Le Lisp 80 version 12.Le manuel de reference

J. Chailloux

IRIA

CENTRE DE ROCQUENCOURT

Original

Rapports Techniques

N° 27

LE LISP 80
VERSION 12
LE MANUEL DE RÉFÉRENCE

Jérôme CHAILLOUX

Juillet 1983

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
tel.(3)954 90 20

I . N . R . I . A .
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay, Cédex

L e L i s p 8 0

v e r s i o n 1 2

L e M a n u e l

d e r é f é r e n c e

Jérôme Chailloux

Mai 1983

Résumé :

ce document est le manuel de référence du système *Le_Lisp 80 (de l'INRIA) version 12*. Ce système, qui s'implante sur toute machine à base d'unité centrale 8080 de Intel ou Z80 de Zilog, sous système CP/M, contient l'interprète, l'éditeur plein écran, et les outils de développement d'un nouveau dialecte du langage Lisp, appelé Le_Lisp.

Abstract :

this document is the reference manual of the system *Le_Lisp 80 (by INRIA) version 12* This system, designed for all the INTEL's 8080 or Zilog80 based computers running the CP/M operating system, contains the interpreter, the Emacs like editor and the developpement tools of a new dialect of the Lisp language, called Le_Lisp.

I . N . R . I . A .
Domaine de Voluceau
Rocquencourt
78153 Le Chesnay, Cédex

L e L i s p 8 0

v e r s i o n 1 2

L e M a n u e l d e r é f é r e n c e

Jérôme CHAILLOUX

Mai 1983

CHAPITRE 1

INTRODUCTION

Le_Lisp 80 est un système Lisp, développé à l'INRIA, qui fonctionne sur des machines à base d'unité centrale de type 8080 d'Intel ou Z80 de Zilog. Ce système contient l'interprète et les outils de développement d'un nouveau dialecte du langage Lisp appelé Le_Lisp, spécialement destiné au développement et à la réalisation de systèmes interactifs de haut niveau axés sur l'enseignement assisté par ordinateur, la conception assistée par ordinateur, la simulation, l'écriture d'interprètes d'autres langages applicatifs et d'une manière générale sur les problèmes d'intelligence artificielle et les domaines qui y sont rattachés.

Du fait de la taille de l'Unité Centrale et de la mémoire utilisée, le système Le_Lisp 80 est une version réduite mais *entièrement compatible* du système Le_Lisp qui existe (ou va exister très prochainement) sur d'autres machines beaucoup plus puissantes en particulier sur :

- 8086/8088 d'Intel sous système CP/M 86 et UNIX V7
- 68000 de Motorola sous système VERSAdos, UNIX V7 et AEGIS.
- Perkin Elmer 32 sous système OS32 et UNIX
- VAX sous système UNIX Berkeley et VMS

Ce manuel est un *manuel de référence* et ne doit pas être considéré comme un manuel d'initiation à Lisp ni comme un manuel de programmation en Lisp. Nous ne saurions que trop conseiller aux tout débutants le livre de C. Queinnec intitulé *LISP, langage d'un autre type,* édité chez Eyrolles 1982, ou bien celui de H. Winston et B. Horn, en anglais, intitulé *LISP* édité chez Addison-Wesley 1981.

Le_Lisp 80 a été écrit en langage machine 8080 par Jérôme Chailloux et Emmanuel Saint-James au cours de l'année 1982.

La version préparatoire de ce manuel de référence a été relue et commentée par Emmanuel Saint-James et Michel Ricart.

TABLE DES MATIERES

1.1 Les différents systèmes Le_Lisp 80

Le_Lisp 80 fonctionne aujourd'hui sur les machines suivantes :

- MICRAL 80-22 G sous système CP/M
- SIL'Z II sous système CP/M
- LOGABAX LX 529 15E sous système CP/M

1.2 Pour débuter avec Le_Lisp 80 sous système CP/M

Pour jouer Le_Lisp 80, il suffit, sous le système CP/M, d'émettre sur le terminal la commande :

```
A>lelisp
```

dans tous les cas, le système répond :

```
****  Le_Lisp 80 (de l'INRIA) jj/mmm/aa  ssssss
.....................................
= Systeme : ssssss  Version : 12  Memoire : xxxx
?
```

dans lequel jj/mmm/aa est la date de la dernière modification de l'interprète, ssssss son type et xxxx le nombre de CONS disponibles. Le système rentre alors dans la boucle principale de l'interprète qui va, indéfiniment, lire une expression sur le terminal, l'évaluer, puis imprimer la valeur de cette évaluation. Le système indique qu'il attend la lecture d'une expression en imprimant, sur le terminal, le caractère ? au début de chaque ligne. La valeur de l'évaluation est imprimée précédée du caractère =.

Voici un exemple de session réalisée avec Le_Lisp 80 sur MICRAL :

```
A>lelisp

**** Le_Lisp 80  (de l'INRIA) 1/Mai/1983 MICRAL
.....................................
= Systeme : MICRAL  Version : 12  Memoire : 4395
? ()
= NIL

? (version)      ; indique le numero de version du systeme
= 12
```

```
? (system)       ; indique le type du systeme utilise
= MICRAL

? (length (oblist))    ; demande le nombre de
= 373                  ; tous les symboles pre-definis

? (+ 1 2 3 4)
= 10

? (* (+ 1 2) (- 6 3) 2)
= 18

? (de fib (n)
?     (cond ((= n 1) 1)
?           ((= n 2) 1)
?           (t (+ (fib (1- n)) (fib (- n 2))))))
= FIB

? (fib 20)
= 6765              ; en moins de 26 secondes !
                    ; sur Z80 a 3 MHz !

? (hanoi 4)         ; pour voir de jolies choses ...
= 4

? (end)
  Que Le_Lisp soit avec vous.

A>                  ; dit CP/M
```

1.3 Quelques trucs à savoir

Enfin voici quelques trucs en vrac, ils sont développés longuement dans les chapitres suivants, à savoir pour utiliser le système confortablement dès le début :

- pour effacer un caractère, utilisez la touche DELETE ou BACKSPACE ou flèche à gauche sur votre clavier; le caractère à gauche du curseur doit s'effacer de l'écran.

- pour détruire une ligne, utilisez le caractère ^X (émis en appuyant simultanément sur les touches CTRL ou CONTROL et X) ou ^U. Un caractère ^X (resp. ^U) est imprimé, un saut de ligne est effectué et le système se remet à l'écoute d'une nouvelle ligne en le signalant par |.

- pour revenir à la boucle principale de l'interprète alors que vous êtes en train de taper une expression et donc que le système fait l'echo de ce que tapez, provoquez une erreur de lecture en tapant deux points l'un à la suite de l'autre. Si rien de visible ne se produit, vous êtes vraisemblablement à l'intérieur d'un commentaire ou d'un symbole spécial; tapez un RETURN de nouveau suivi de deux points.

- pour revenir à la boucle principale de l'interprète si votre programme boucle, tapez ^B suivi de RETURN. Le système est prêt à lire une nouvelle expression.

- pour revenir au système d'exploitation, tapez ^C suivi de RETURN.

- pour charger un fichier précédemment créé, dîtes simplement :

```
? % fichier
```

c'est-à-dire le caractère % suivi du nom du fichier sans son extension (qui est .LL par défaut sous système CP/M).

- pour créer ou éditer un fichier, appelez l'éditeur PEPE en évaluant la forme :

```
? & fichier
```

cet éditeur, intégré dans le système Le_Lisp 80, se veut le plus proche possible du célèbre éditeur Emacs en particulier au niveau de la sémantique attachées aux clés. Pour visualiser un récapitulatif des commandes, utilisez, quand vous êtes sous l'éditeur, la commande :

```
[esc] ?        ou bien
[sortie] ?
```

c'est-à-dire en entrant le caractère [esc] ou [sortie] suivi du caractère ?

- pour insérer un commentaire dans une expression Lisp, tapez un point-virgule ; tout le reste de la ligne est ignoré jusqu'à la fin complète de la ligne.

Enfin le source d'un grand nombre d'utilitaires, écrits en Lisp, est disponible : éditeur, paragrapheur, aides à la mise au point, démonstration video ... Lisez-les, comprenez-les et n'hésitez pas à les étendre et à les améliorer.

CHAPITRE 2

L'INTERPRETE

2.1 Les Objets de base

L'interprète Le_Lisp 80 permet de manipuler des objets nommés S-expressions (pour *Symbolic expressions).*

Ces objets sont classés en 3 types distincts :

- les symboles
- les nombres
- les listes

Pour des raisons d'encombrement mémoire, Le_Lisp 80 ne dispose pas de chaînes de caractères ni de tableaux.

Toute S-expression est représentée en machine au moyen d'un pointeur (ou d'une adresse) vers sa ou ses valeurs. L'accès aux valeurs des différents objets est donc toujours effectué au moyen d'une indirection. L'interprète Le_Lisp est donc spécialisé dans la manipulation de pointeurs.

Le_Lisp 80 fonctionne avec une Unité Centrale à 8 bits. L'espace adresse y est de 16 bits. La mémoire adressable est donc de 64k octets.

Pendant le fonctionnement du système cette mémoire va contenir :
- le noyau résident du système d'exploitation (e.g. CP/M)
- l'interprète Le_Lisp 80
- la pile de donnée et de contrôle de cet interprète
- la zone allouée aux symboles
- la zone allouée aux listes

2.1.1 Les symboles

Ils jouent le rôle d'identificateurs et servent à nommer les variables et les fonctions. Ils sont créés implicitement dès leur lecture dans le flux d'entrée ou explicitement par les fonctions IMPLODE et CONCAT ; nul besoin donc de les déclarer.

Leur nom externe (Print name ou P-NAME) est une suite de caractères quelconques (contenant au moins un caractère non numérique) de longeur limitée à 62 caractères. On peut insérer dans un P-NAME des caractères délimiteurs s'ils sont précédés du caractère / *(slash)* ou mieux, si le P-NAME contient des caractères spéciaux, on peut encadrer tout le P-NAME avec le caractère | *valeur absolue* (voir la section sur la lecture standard).

Un symbole est représenté dans l'interprète par un pointeur sur un descriptif stocké dans une zone spéciale de la mémoire.
Ce descriptif est constitué des 7 propriétés naturelles suivantes :

C-VAL (abréviation de Cell-VALue) qui contient à tout moment la valeur associée au symbole considéré comme une variable. L'accès à cette valeur est extrêmement rapide. A la création d'un symbole, sa C-val est *indéfinie.* Toute tentative de consultation d'un symbole qui n'a pas encore reçu de valeur provoque irrémédiablement une erreur. L'accès à la C-VAL peut également s'effectuer au moyen de la fonction CVAL.

P-LIST (abréviation de Properties LIST) qui contient à tout moment la liste des propriétés du symbole. Ces propriétés sont gérées par l'utilisateur au moyen des fonctions spéciales sur les P-LIST (ADDPROP, PUTPROP, GETPROP, REMPROP). Le système n'utilise jamais les P-LIST des symboles pour ses besoins propres.

F-VAL (abréviation de Function-VALue) qui contient à tout moment la valeur associée au symbole considéré comme une fonction. Cette valeur peut-être :
- une adresse machine dans le cas des SUBR
- une S-expression dans le cas des EXPR

L'accès à la F-VAL est réalisé au moyen des fonctions VALFN, REMFN et GETDEF.

F-TYPE (abréviation de Function TYPE) qui contient le type de la fonction stockée dans la F-VAL. L'ensemble F-VAL, F-TYPE permet à l'interprète de lancer les fonctions d'un manière extrêmement rapide. La consultation, en clair, du F-TYPE des symboles est réalisée au moyen de la fonction TYPEFN.

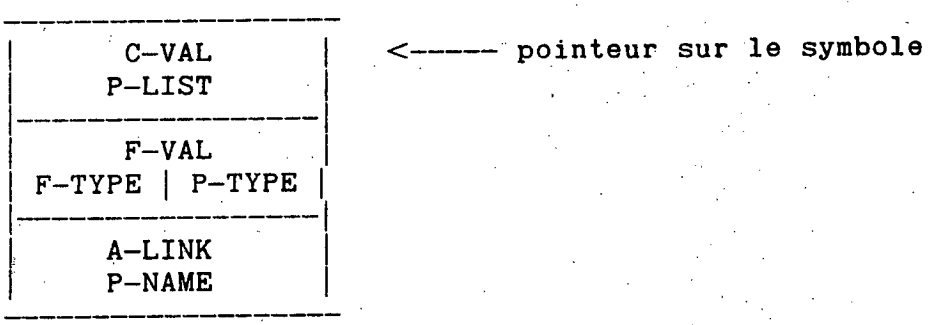
P-TYPE (abréviation de Print TYPE) qui contient les informations nécessaires à l'édition de la représentation externe du symbole
- comme variable : en indiquant la restitution du caractère *valeur absolue* pour encadrer les caractères du P-NAME.
- comme fonction : pour permettre au PRETTY-PRINT de connaître le

format d'édition à utiliser. L'accès au P-TYPE d'un symbole est effectué au moyen de la fonction spéciale PTYPE.

A-LINK (abréviation de Atom-LINK) qui contient l'adresse de l'atome suivant dans la table des symboles. Ce lien permet entre autre de gérer facilement le hachage (hash-coding) de la table des symboles. Cet attribut n'est pas accessible à l'utilisateur directement.

P-NAME (abréviation de Print-NAME) qui contient l'adresse de la chaîne de caractères représentant le nom du symbole.

Ces propriétés naturelles sont rangées en mémoire suivant le schéma :



Certains symboles sont déjà connus à l'appel du système :
- les constantes littérales (qui contiennent leur propre adresse en C-VAL) comme : NIL T LAMBDA FLAMBDA MLAMBDA SUBR NSUBR FSUBR EXPR FEXPR MACRO QUOTE ...
- les fonctions prédéfinies.

2.1.2 Les nombres

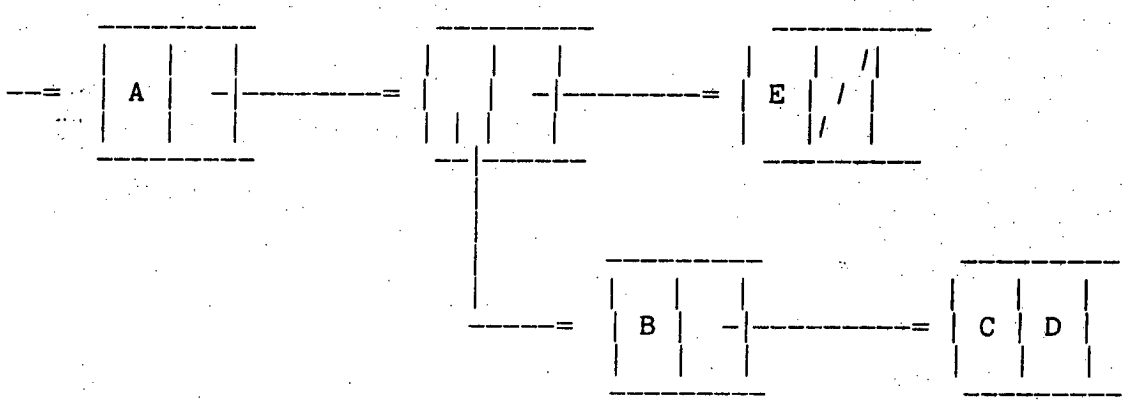
Le_Lisp manipule des nombres entiers sur 14 bits (permettant de calculer dans l'intervalle : [-2^13 ,, +2^13-1], c'est-à-dire [-8192 ,, +8191]).

Le P-NAME d'un nombre est la représentation de sa valeur dans la base de conversion courante (voir la fonction OBASE).
La valeur d'un nombre est ce nombre lui-même. Un nombre n'a ni C-VAL ni P-LIST.

2.1.3 Les listes

Les listes sont représentées d'une manière standard : les différents éléments d'une liste sont stockés dans une cellule de liste (constituée d'un doublet de pointeurs) dont la partie gauche (le CAR) contient l'élément, et la partie droite (le CDR) contient un pointeur sur l'élément suivant ou NIL pour le dernier élément de la liste.

Ex : la liste (A (B C . D) E) est stockée en mémoire sous la forme :



Attention : Le_Lisp 80 n'autorise pas la création de doublets de listes dont le CDR est un nombre. Si cela se produit, l'erreur ERNUM se déclenche.

2.2 Le fonctionnement de base de l'interprète

2.2.1 L'évaluation d'un atome

La valeur d'un symbole (considéré comme une variable) est sa C-VAL. L'évaluation d'un symbole dont la C-VAL est indéfinie (c'est à dire auquel on n'a pas encore donné de valeur) provoque lors de son évaluation l'erreur ERUDV dont le libellé est :

```
** <fn> : variable indefinie : <sym>
```

dans lequel le nom du symbole incriminé <sym> est imprimé ainsi que le nom de la fonction <fn> qui a provoqué l'erreur (en général la fonction évaluateur EVAL).

La valeur d'un nombre est ce nombre, nul besoin donc de les *quoter*.

2.2.2 L'évaluation d'une liste

L'évaluateur considère toujours une liste comme un appel de fonction. Cette liste s'appelle une forme. Le CAR de cette forme est la fonction, le CDR de la forme les arguments de la fonction.
La valeur d'une forme est la valeur retournée par l'application de la fonction aux arguments.

2.3 Les fonctions

Une fonction (le CAR d'une forme) peut être n'importe quelle S-expression. Le CDR de la forme est la liste des paramètres actuels de la fonction.

Si la fonction est un symbole, la fonction à utiliser est celle qui a été associée à ce symbole
- soit à l'initialisation du système (c'est le cas des fonctions prédéfinies qui sont appelées également fonctions standard)
- soit par l'utilisateur au moyen des fonctions de définition statiques ou dynamiques.

Si aucune fonction n'a été associée à ce symbole, une indirection est opérée sur la C-VAL (c'est-à-dire sur la valeur du symbole en tant que variable). Si cette C-VAL est indéfinie ou contient une constante littérale, l'erreur ERUDF se déclenche dont le libellé est :

```
** <fn> : fonction indefinie : <sym>
```

dans lequel le nom du symbole incriminé <sym> est imprimé ainsi que le nom de la fonction <fn> ayant provoqué l'erreur (en général l'une des deux fonctions interprètes EVAL ou APPLY).

```
ex : (CONS 'A 'B)          ->  (A . B)
     (SETQ KONS 'CONS)     ->  CONS
     (KONS 'X 'Y)          ->  (X . Y)
```

Dans le cas où la fonction est un nombre, l'interprète provoque également une erreur dont le libellé est le même que dans le cas précédent.

```
ex : (3 '(1 2 3))  ->
     ** EVAL : fonction indefinie : 3
```

Dans le cas où la fonction est une liste, il peut s'agir :

- d'une déclaration explicite de fonction. Dans ce cas le premier élément de la liste doit être l'atome spécial LAMBDA, MLAMBDA ou FLAMBDA.

```
ex : ((LAMBDA (x) (+ x x 2)) 5)    ->  12
     ((FLAMBDA (x) x) (+ x x))     -> (+ x x)
```

- d'une nouvelle forme qui doit être évaluée et dont la valeur est la fonction à utiliser. On a donc affaire dans ce cas non pas à des fonctions constantes (anonymes ou non) mais à des fonctions variables. Lisp est l'un des rares langages dans lequel il est possible d'écrire des appels très agréables du genre :

```
((IF (< n 0) '* '+) val 2)
```

Le_Lisp possède 5 familles de fonctions qui vont être traitées différemment par l'interprète :

- les SUBR et les FSUBR écrites en langage machine.
- les EXPR, les FEXPR et les MACRO écrites en Lisp.

2.3.1 Les SUBR

Les SUBR sont des fonctions écrites en langage machine, résidentes dans l'interprète dès son lancement (les fonctions prédéfinies). L'exécution de ces fonctions est extrêmement rapide. Ces fonctions possèdent des arguments qui sont toujours évalués. Il existe des SUBR à nombre fixe d'arguments (en général ce nombre est plus petit que 4) et des SUBR à nombre variable d'arguments (ces dernières sont parfois appelées NSUBR). Pour les SUBR à nombre fixe d'arguments l'interprète teste si le bon nombre d'arguments est fourni à l'appel. S'il n'y a pas assez d'arguments à l'appel, les arguments manquants sont mis, par défaut, à la valeur NIL. S'il y a trop d'arguments, l'erreur ERWNA se déclenche dont le libellé est :

```
** <fn> : trop d'arguments : <s>
```

dans lequel le nom de la SUBR incriminée <fn> est imprimé ainsi que le reste des arguments <s> non évalué.

Pour les NSUBR l'interprète teste parfois le nombre minimum d'arguments à l'appel et peut provoquer une erreur identique à celle des SUBR.

Il est possible de rajouter des fonctions de ce type, en les écrivant directement en langage machine (voir la section sur les fonctions d'accès au système) mais c'est un sport toujours très périlleux.

Ces fonctions sont très nombreuses dans l'interprète standard, entre 200 et 300 en fonction du système utilisé.

```
Ex : (CONS 1)        =>  (1)
     (CONS 1 'B 3)   =>  ** CONS : trop d'arguments : (3)
     (CONS 1 'B)     =>  (1 . B)
```

2.3.2 Les FSUBR

Les FSUBR sont également des fonctions écrites en langage machine (donc exécutées extrêmement rapidement), résidentes dans l'interprète dès son lancement. Ces fonctions possèdent des arguments en nombre variable qui ne sont jamais évalués par l'interprète lui-même. Ces fonctions, très spéciales, autrefois appelées formes spéciales, sont principalement utilisées comme fonctions de contrôle de l'interprète ou comme fonctions de manipulation de noms. Elles sont peu nombreuses.

2.3.3 Les EXPR

Les EXPR sont des fonctions écrites en Lisp et interprétées par les fonctions standard d'évaluation (EVAL et APPLY). Ces fonctions possèdent un nombre quelconque de paramètres formels, représentés par une liste de variables <lvar> ainsi qu'un corps de fonction constitué d'un nombre quelconque d'expressions <s1> ... <sN>.

On décrit une fonction de ce type en utilisant une liste, appelée LAMBDA-expression, de la forme :

(LAMBDA <lvar> <s1> ... <sN>)

dans laquelle le symbole LAMBDA est un indicateur de fonction de type EXPR, <lvar> est la liste des paramètres formels et <s1> ... <sN> le corps de la fonction.

```
ex : (LAMBDA (x y) (CONS (CAR x) (CDR y)))
```

La définition des fonctions de type EXPR (c'est-à-dire l'association d'une LAMBDA-expression à un symbole) est réalisée au moyen de la fonction DE.

L'évaluation d'un appel de fonction de type EXPR s'opère en 3 étapes :

1 - liaison des paramètres actuels aux paramètres formels. Cette liaison est réalisée en Lisp après avoir sauvé les anciennes valeurs des paramètres formels dans la pile.
2 - évaluation des différentes expressions du corps <s1> ... <sN>. La valeur retournée par l'appel de la fonction sera la valeur de la dernière évaluation (c'est-à-dire celle de <sN>)
3 - destruction des liaisons effectuées en 1 - et restitution des anciennes valeurs sauvées dans la pile.

Le type de liaison des paramètres est fonction de la forme de la liste des paramètres formels <lvar>.

- dans le cas le plus simple, <lvar> est une liste de variables. La liaison s'effectue élément par élément entre la liste des variables (paramètres formels) et la liste des

valeurs (paramètres actuels évalués). Si la liste des paramètres actuels est plus longue que celle des paramètres formels, il se produit l'erreur ERWNA comme pour les SUBR. Si la liste des paramètres formels est plus longue que celle des paramètres actuels, les variables restantes sont liées à la valeur NIL par défaut.

- dans le deuxième cas, <lvar> est une variable simple. Tous les paramètres actuels sont alors évalués puis rassemblés en une liste qui est liée à la variable. On appelle souvent ce type de fonction des LEXPR (terminologie Maclispienne).

- enfin dans le dernier cas, qui est la généralisation des deux premiers cas, <lvar> est un arbre quelconque. La liaison va s'effectuer, au moyen d'une procédure récursive, entre tous les éléments de l'arbre des variables et les éléments de l'arbre des valeurs.

Si la liste des arguments est mal-formée, l'erreur ERWLA se déclenche dont le libellé est :

```
** <fn> : liaison illegale : <s>
```

dans lequel <fn> est le nom de la fonction ayant déclenchée l'erreur (EVAL ou APPLY) et <s> la liste d'arguments défectueuse.

Voici, décrit en Lisp ce mécanisme de liaison généralisé :

```
(DE BINDVAR (lvar larg)
    ; lvar : liste (ou arbre) des variables
    ; larg : liste des arguments
    (IF (SYMBOLP lvar)
        (IFN  lvar
              (WHEN larg (SYSERROR 'EVAL 'ERWNA larg))
              (PUSH (CVAL lvar))
              (PUSH lvar)
              (SET lvar larg))
        (IFN (LISTP larg)
             (SYSERROR 'EVAL 'ERWLA (LIST lvar larg))
             (BINDVAR (CAR lvar) (CAR larg))
             (BINDVAR (CDR lvar) (CDR larg)))))
```

Exemples de liaison des EXPR

```
((LAMBDA (x y z) (LIST x y z))
    (PRIN 1) (PRIN 2) (PRIN 3))
123   et  ->  (1 2 3)

((LAMBDA (x y z) (LIST x y z))
    (PRIN 1))
1  et  ->  (1 NIL NIL)

((LAMBDA (x y z) (LIST x y z)) 1 2 3 4)
** EVAL : trop d'arguments : (4)
```

```
((LAMBDA x x) (PRIN 1) (PRIN 2) (PRIN 3))
123  et  ->  (1 2 3)

((LAMBDA (x y . z) (LIST x y z))
    (PRIN 1) (PRIN 2) (PRIN 3) (PRIN 4))
1234  et  ->  (1 2 (3 4))

((LAMBDA ((x . y) . z) (LIST x y z)) '(A . B) 'C))
     -> (A B (C))

((LAMBDA (x (y . z)) (list x y z)) 'a 'b)
** EVAL : liaison illegale : ((y . z) b)
```

2.3.4 Les FEXPR

Les FEXPR sont des fonctions écrites en Lisp et interprétées par les fonctions standard d'évaluation (EVAL et APPLY). Tout comme les EXPR ces fonctions possèdent un nombre quelconque de paramètres formels, qui sont rangés dans une liste <lvar> ainsi qu'un corps de fonction constitué d'un nombre quelconque d'expressions <s1> ... <sN>.

On décrit une fonction de ce type en utilisant une liste, appelée FLAMBDA-expression, de la forme :

(FLAMBDA <lvar> <s1> ... <sN>)

dans laquelle le symbole FLAMBDA est un indicateur de fonction de type FEXPR, <lvar> est la liste des paramètres formels et <s1> ... <sN> le corps de la fonction.

```
ex : (FLAMBDA (var val) (SET var (EVAL val)))
```

La définition des fonctions de type FEXPR (c'est-à-dire l'association d'une FLAMBDA-expression à un symbole) est réalisée au moyen de la fonction DF.

Ces fonctions ne diffèrent des EXPR que par le mode de liaison des paramètres car c'est l'ensemble de paramètres actuels *non évalués,* c'est-à-dire la CDR de la forme elle-même, qui est lié à la liste des variables <lvar>.

2.3.5 Les MACRO

L'interprète reconnait un autre type de fonctions, les MACRO. Tout comme les EXPR et les FEXPR, ces fonctions écrites en Lisp possèdent un nombre quelconque de paramètres formels, qui sont rangés dans une liste <lvar> ainsi qu'un corps de fonction constitué d'un nombre quelconque d'expressions <s1> ... <sN>.

On décrit une fonction de ce type en utilisant une liste, appelée MLAMBDA-expression, de la forme :

(MLAMBDA <lvar> <s1> ... <sN>)

dans laquelle le symbole MLAMBDA est un indicateur de fonction de type MACRO, <lvar> est la liste des paramètres formels et <s1> ... <sN> le corps de la fonction.

```
ex : (MLAMBDA (n var) ['SETQ var ['CDR var]])
```

La définition des fonctions de type MACRO (c'est-à-dire l'association d'une MLAMBDA-expression à un symbole) est réalisée au moyen de la fonction DM.

Pour évaluer une forme dont la fonction est une MACRO, l'interprète évalue tout d'abord la fonction associée à cette MACRO avec toute la forme (évidemment non évaluée) comme paramètre actuel puis re-évalue la valeur retournée de cette première évaluation. L'évaluation d'une macro se fait donc en deux temps.
C'est l'appel de la macro tout entier qui est pris comme paramètre actuel. Il est donc possible de réaliser des modifications physiques de la forme elle-même à la première évaluation de la macro.

L'utilisation des macros, véritable sport en soit, permet d'étendre le langage très facilement et très puissamment, Lisp servant à la fois de langage de description et d'expansion des macros.

2.4 La définition des fonctions

Il existe deux types d'utilisation des fonctions :

- une utilisation statique : les fonctions sont définies d'une manière globale et gardent leurs définitions tant qu'on ne les modifient pas explicitement. Ce type de définition ne permet pas de conserver les définitions antérieures (voir les fonctions DE/DF/DM)

- une utilisation dynamique : les fonctions peuvent changer de définition durant certaines évaluations et retrouver par la suite leurs anciennes définitions (voir la fonction FLET).

Ces deux types d'utilisation s'appliquent aussi bien aux fonctions nommées (associées à un symbole) qu'aux fonctions anonymes (les lambda-expressions), de n'importe quel type.

2.5 L'évaluateur

L'évaluateur Le_Lisp (i.e. la fonction interprète EVAL) a été spécialement étudié du point de vue de la vitesse d'exécution.

Toutes les fonctions atomiques sont lancées très rapidement au moyen d'un branchement indirect indexé (par la F-VAL et le F-TYPE).

L'évaluateur minimise le nombre de CONS internes, en n'utilisant jamais la fonction EVLIS pour ses besoins propres (pour évaluer les arguments des lambda-expressions et des NSUBR) sauf en cas de demande explicite au moyen d'une fonction de type LEXPR.

Enfin, l'interprète réduit la consommation de pile de l'utilisateur : dans le cas général, lorsqu'une fonction non standard est activée, les anciennes valeurs des paramètres sont sauvegardées dans la pile, puis restaurées une fois le calcul terminé. L'évaluateur reconnaît de nombreux cas, appelés *récursivités terminales,* où ces valeurs ne seront plus utilisées, et n'effectue alors aucune sauvegarde. Cette reconnaissance conduit le plus souvent à une interprétation totalement itérative des définitions récursives. Ces transformations et l'absence d'appel à EVLIS autorise, sans perte d'efficacité, une programmation entièrement applicative que nous ne saurions trop recommander.

Voici, rassemblée et commentée par Emmanuel Saint-James, la liste des cas de récursivités terminales reconnues et traitées par l'interprète Le_Lisp 80.

Mais auparavant une définition :

Soient S une expression LISP, S' une sous-expression de S. S' sera dite terminale dans S si S = S' ou si S' est la dernière expression à être évaluée dans une autre sous-expression S", elle-même terminale dans S. Dans ce dernier cas, S" a la forme (f ... S' ...). Nous appelerons f la *fonction enveloppe* de S (quelque soit son F-type).

Le type et le nombre des enveloppes déterminent les différents cas de récursivités terminales.

Cas 1 : la situation la plus simple, c'est à dire sans enveloppe, est fournie par les boucles système :

```
(DE TOPLEVEL (DER)
    (TOPLEVEL (PRINT (EVAL (READ)))))
```

Une seule sauvegarde de la valeur de DER sera effectuée quelque soit le nombre d'appel, ce qui transforme cette récursion infinie en une boucle infinie, indépendante de la taille de la pile.

Cas 2 : cette propriété n'est pas l'apanage des récursions infinies. Voici un exemple avec une seule enveloppe, permettant l'arrêt de la récursion:

```
(DE LAST (L)
    (IF (CDR L)
        (LAST (CDR L))
        (CAR L)))
```

Cas 3 : ces interprétations itératives sont indépendantes des structures de contrôle employées (IF, IFN, COND, OR, AND, UNLESS, WHEN, SELECTQ, PROGN) et de leur profondeur d'imbrication. En voici un exemple : la recherche d'une sous-liste dans une liste, où un appel non terminal précède un appel terminal (noter la nécessité de la sauvegarde, pour le premier appel) :

```
(DE OCCUR (E L)
    (OR (EQUAL E L)
        (AND (CONSP L)
             (OR (OCCUR E (CAR L))
                 (OCCUR E (CDR L))))))
```

Cas 4 : l'interprétation itérative a également lieu lorsque plusieurs fonctions travaillent en récursivités terminales croisées. Dans cette situation, la valeur de chacun des paramètres ne sera sauvegardée qu'une fois, quelques soient les cheminements à travers le graphe d'appel. Cette propriété permet une écriture très aisée d'automates de langages rationnels ; en voici un qui reconnait les mots ayant un nombre pair de fois la lettre "A" et un nombre impair de fois la lettre "B", indépendamment des autres lettres:

```
(DE PP (PP)
    (AND PP
         (SELECTQ (CAR PP)
             (A  (IP (CDR PP)))
             (B  (PI (CDR PP)))
             (T  (PP (CDR PP))))))
```

```
(DE IP (IP)
    (AND IP
         (SELECTQ (CAR IP)
             (A  (PP (CDR IP)))
             (B  (II (CDR IP)))
             (T  (IP (CDR IP))))))

(DE PI (PI)
    (OR PI
        (SELECTQ (CAR PI)
             (A  (II (CDR PI)))
             (B  (PP (CDR PI)))
             (T  (PI (CDR PI))))))

(DE II (II)
    (AND II
         (SELECTQ (CAR II)
             (A  (PI (CDR II)))
             (B  (IP (CDR II)))
             (T  (II (CDR II))))))
```

Cas 5 : l'interprétation itérative est également assurée en cas d'imbrication avec les autres cas de sauvegarde (BIND, FLET) et les déclarations d'étiquettes d'échappement (TAG), car la récursivité terminale est aussi repérée pour ces fonctions. Voici par exemple une boucle système qui récupère les erreurs sans imprimer de messages, et se signale par un prompt particulier:

```
(DE BOTTOMLEVEL (DER)
    (TAG MERROR
         (FLET ((SYSERROR X (EXIT MERROR)))
               (BIND ((PROMPT '| |))
                     (BOTTOMLEVEL
                         (PRINT (EVAL (READ)))))))))
```

Cas 6 : enfin, la détection d'une récursivité terminale a lieu lors de l'évaluation de la valeur d'un échappement. L'utilisation de cette propriété permet d'éviter deux évaluations d'un calcul (ou la liaison de son résultat à une variable) : l'écriture d'une conditionnelle peut se faire en commençant par la clause "vrai" dans laquelle le test est réalisé, et un échappement provoqué en cas d'echec. En voici un exemple: il s'agit de trouver dans une liste de A-listes (lal), celle contenant la liaison d'une certaine variable (v) avec une valeur différente d'une valeur donnée (n).

```
(DE LSA (lal v n)
    (TAG next
         (AND lal
              (NEQ v
                   (CDR (OR (ASSQ n (CAR lal))
                            (EXIT next
                                  (LSA (CDR lal) v n)))))
              (CAR lal))))
```

On n'aura pas manqué de noter que toutes les enveloppes ci-dessus étaient des FSUBRs. Dans le cas contraire, les sauvegardes ne seront là non plus pas faites, mais la présence dans la pile, de l'adresse de la SUBR et des valeurs de ses premiers arguments, fera que l'interprétation ne sera plus totalement itérative.

Cas 7 : une 1SUBR comme enveloppe :

```
(DE LENGTH (l)
    (IF (ATOM l)
        0
        (1+ (LENGTH (CDR l)))))
```

Cas 8 : une 2SUBR :

```
(DE MAPCAR2 (f l)
    (WHEN (CONSP l)
          (CONS (FUNCALL f (CAR l))
                (MAPCAR f (CDR l)))))
```

Cas 9 : une 3SUBR :

```
(DE PAIRLIS (ln lv)
    (WHEN (CONSP ln)
          (ACONS (CAR ln)
                 (CAR lv)
                 (PAIRLIS (CDR ln) (CDR lv)))))
```

Cas 10 : l'absence de sauvegarde a également lieu lorsque la fonction enveloppe est la fonction appelée, elle même en position terminale. Cette situation, que nous appellerons *récursivité auto–enveloppée* est fréquente lorsqu'une fonction à deux appels récursifs doit être partiellement dérécursivée, afin d'accèder au résultat intermédiaire. La fonction suivante par exemple, établit la nomenclature des atomes apparaissant dans une liste, et cela sans répétitions :

```
(DE NOMEN (e r)
    (COND ((CONSP e)
                (NOMEN (CAR e) (NOMEN (CDR e) r)))
          ((NULL e) r)
          ((MEMQ e r) r)
          (T (CONS e r))))
```

Cas 11 : naturellement, ces interprétations sont de nouveau indépendantes de la profondeur d'imbrication des enveloppes et de leurs types. Voici un exemple où se mêlent récursivités croisées et auto-enveloppées; les valeurs des paramètres de la deuxième fonction ne seront sauvegardées qu'une fois. Il s'agit de nouveau d'une nomenclature : celle des variables apparaissant dans une expression, afin d'en engender leur fermeture ; il convient de ne pas y inclure le nom des fonctions, d'où la nécessité de la deuxième fonction.

```
(DE VARLIB (el rl)
    (COND  ((CONSP el) (ARGLIB (CDR el) rl))
           ((CONSTANTP el) rl)
```

```
                ((MEMQ e1 r1) r1)
                (T (CONS e1 r1))))

(DE ARGLIB (e2 r2)
    (IFN e2
         r2
         (ARGLIB (CDR e2)
                 (VARLIB (CAR e2) r2))))
```

Cas 12 : le cas de certaines NSUBRs (APPEND, NCONC, MCONS, CONCAT, MIN, MAX, *, +, TYO) en enveloppe est particulièrement intéressant : l'interprète transforme les deux appels successifs de la NSUBR en un seul appel avec deux fois plus d'arguments. L'exemple suivant est une transformation d'une A-liste en P-liste :

```
(DE PLENAL (l)
    (WHEN (CONSP l)
          (MCONS (CAAR l)
                 (CDAR l)
                 (PLENAL (CDDR l)))))
```

Pour donner une idée qualitative de ce dernier type de transformation, faisons remarquer que la définition récursive classique de factorielle, sera automatiquement transformée en l'application de la multiplication n-aire sur la génération de l'intervalle [1,n]. Cette économie d'espace étant plus importante que les précédentes, on distingue les deux cas en qualifiant l'interprétation de *semi–itérative* dans le cas des NSUBRs, et *semi–récursive* dans les autres cas.

Nous espérons que ce panorama aura convaincu le lecteur de ce que la récursivité n'est plus synonyme d'inefficacité.

2.6 La définition méta–circulaire de l'interprète

Enfin voici une description méta-circulaire de l'interprète Le_Lisp. Cette description permet d'avoir une vue globale du fonctionnement de l'interprète en particulier de la liaison des arguments et le traitement des erreurs mais ne représente pas la véritable mise en œuvre qui est beaucoup plus performante notamment au niveau du nombre de CONS réalisés dans les fonctions de l'interprète et au niveau de l'encombrement de la pile : l'interprète simule les appels de la fonction EVLIS à l'aide de la pile, et n'effectue des sauvegardes d'environnement que si nécessaire.

```
; boucle principale
(DE TOP-LEVEL ()
    (TAG SYSERROR
         (PRINT '|Toplevel|)
         (SETQ IT (LET ((STACK)) (EVAL (READ))))
         (PRINT '|= | IT))
    (TOP-LEVEL))


; traitement des erreurs de l'interprète
(DE SYSERROR (f m a)
    (PRINT '|** | f
           '| : | (IF (AND (SYMBOLP m)
                            (BOUNDP m))
                       (CVAL m)
                       m)
           '| : | a)
    (BREAK))


; les libellés des erreurs
(SETQ
    ERSXT   '|erreur de syntaxe|
    ERIOS   '|erreur disque|
    EROOB   '|argument hors-limite|
    ERUDV   '|variable indefinie|
    ERUDF   '|fonction indefinie|
    ERUDT   '|echappement indefini|
    ERWNA   '|trop d'arguments|
    ERWLA   '|liaison illegale|
    ERNUM   '|CDR numerique|
    ERNNA   '|l'argument n'est pas un nombre|
    ERNV    '|l'argument n'est pas une variable|
    ERNAA   '|l'argument n'est pas un symbole|
    ERNLA   '|l'argument n'est pas un CONS|

    FERAT   '|zone des symboles pleine|
    FERLS   '|zone des listes pleine|
    FERFS   '|zone pile pleine|)
```

```
; pour réaliser des mises au point
(DE BREAK ()
    (EXIT SYSERROR 'SYSERROR))


; le traitement des caractères d'interruption
(DE ITEVAL (n)
    (SELECTQ n
        (#^ B (BREAK))
        (#^ C (END))
        (#^ S (TYI))
        (T ())))

;-----   L'évaluateur proprement dit

; évaluation d'une forme quelconque
(DE EVAL (forme)
    (COND
       ((NUMBERP forme) forme)
       ((CONSP forme)
              (EVALFONC (CAR forme)  (CDR forme)))
       (T (IF (BOUNDP forme)
              (CVAL forme)
              (SYSERROR 'EVAL 'ERUDV forme)))))


; évalue une liste
(DE EVALFONC (fonct larg)
    (COND
        ((NUMBERP fonct)
            (SYSERROR 'EVAL 'ERUDF fonct))
        ((SYMBOLP fonct)
            (IF (TYPEFN fonct)
                (EVALINTERNAL (TYPEFN fonct) (VALFN fonct) larg)
                (IF (OR (CONSTANTP (CVAL fonct))
                        (EQ fonct (CVAL fonct)))
                    (SYSERROR 'EVAL 'ERUDF fonct)
                    (EVALFONC (CVAL fonct) larg)))))
        ((EQ (CAR fonct) 'LAMBDA)
            (EVALINTERNAL 'EXPR (CDR fonct) larg))
        (T  (EVALFONC (EVAL fonct) larg))))


; évaluation d'une fonction suivant son type
(DE EVALINTERNAL (ftype fval larg)
    (SELECTQ ftype
      (0SUBR  (CALL fval () () ()))
      (1SUBR  (CALL fval (EVAL (CAR larg)) () ()))
      (2SUBR  (CALL fval
                    (EVAL (CAR larg))
                    (EVAL (CADR larg)
```

```
                    ())))
        (3SUBR  (CALL fval
                      (EVAL (CAR larg))
                      (EVAL (CADR larg))
                      (EVAL (CADDR larg))))
        (NSUBR (CALLN fval (EVLIS larg)))
        (FSUBR (CALL fval larg))
        (EXPR  (EVALAMBDA fval (EVLIS larg)))
        (FEXPR (EVALAMBDA fval larg))
        (MACRO (EVAL (EVALAMBDA fval forme)))
        (T     ()))


; construit la liste des valeurs des évaluations
; de tous les éléments de l
(DE EVLIS (l)
    (IF (NULL l)
        ()
        (CONS (EVAL (CAR l)) (EVLIS (CDR l)))))


; évalue le corps l
(DE EPROGN (l)
    (IF (NULL (CDR l))
        (EVAL (CAR l))
        (EVAL (CAR l))
        (EPROGN (CDR l))))


; applique une F-VAL de type : (<lvar> <s1> ... <sN>)
; à la liste d'arguments larg
(DE EVALAMBDA (f larg)
    (PUSH ())           ; () : marqueur de fin de sauvegarde
    (BINDVAR (CAR f) larg)
    (PROTECT (EPROGN (CDR f))
             (UNBINDVAR)))


; effectue la liaison d'arbre généralisée
(DE BINDVAR (lvar larg)
    (IF (ATOM lvar)
        (IFN  lvar
              (WHEN larg (SYSERROR 'EVAL 'ERWNA larg))
              (PUSH (CVAL lvar))
              (PUSH lvar)
              (SET lvar larg))
        (IFN (LISTP larg)
             (SYSERROR 'EVAL 'ERWLA (LIST lvar larg))
             (BINDVAR (CAR lvar) (CAR larg))
             (BINDVAR (CDR lvar) (CDR larg)))))
```

```
; restaure les anciennes valeurs
(DE UNBINDVAR ()
    (UNTILEXIT DONE
       (SET (OR (POP) (EXIT DONE)) (POP))))


; avec les fonctions de manipulation de pile

(DE PUSH (l) (NEWL stack l))

(DE POP () (NEXTL stack))

; et une pile

(SETQ stack ())

; et pour les fanatiques, un véritable APPLY

; applique la fonction fonct aux arguments larg
(DE APPLY (fonct larg)
    (COND
      ((NUMBERP fonct)
          (SYSERROR 'APPLY 'ERUDF fonct))
      ((SYMBOLP fonct)
          (APPLYINTERNAL (TYPEFN fonct) (VALFN fonct) larg))
      ((EQ (CAR fonct) 'LAMBDA)
          (APPLYINTERNAL 'EXPR fonct larg))
      (T (APPLY (EVAL fonct) larg))))


; application d'une fonction suivant son type
(DE APPLYINTERNAL (ftyp fval larg)
    (SELECTQ ftyp
      (0SUBR (CALL fval () () ()))
      (1SUBR (CALL fval (CAR larg) () ()))
      (2SUBR (CALL fval (CAR larg) (CADR larg) ()))
      (3SUBR (CALL fval (CAR larg) (CADR larg) (CADDR larg)))
      (NSUBR (CALLN fval larg))
      (FSUBR (CALL fval larg () ()))
      (EXPR  (EVALAMBDA fval larg))
      (FEXPR (EVALAMBDA fval larg))
      (MACRO (EVAL (EVALAMBDA fval (CONS fonct larg))))
      (T     (SYSERROR 'APPLY 'ERUDF fval))))
```

CHAPITRE 3

LES FONCTIONS PREDEFINIES

Toutes les fonctions qui vont être décrites dans ce chapitre sont résidentes dans tous les systèmes utilisés.

Pour chacune d'elles on donnera son type (SUBR ou FSUBR) ainsi que le nombre standard d'arguments, et pour chaque argument le type souhaité ou requis, ces types étant notés :

- <s> pour une S-expression quelconque
- <l> pour une liste
- <a> pour un atome quelconque (symbole ou nombre)
- <sym> pour un symbole
- <var> pour un symbole autre que T ou NIL
- <n> pour un nombre
- <ch> pour un caractère (c'est-à-dire un atome dont le P-NAME n'a qu'un caractère)
- <cn> pour le code ASCII d'un caractère
- <fn> pour une fonction (un symbole ou une lambda-expression)

Dans la mesure du possible, les SUBR seront décrites en Lisp sous forme de DE, DF ou DM. Ces descriptions ne sont que les équivalents Lisp de ces fonctions et ne représenteront que leurs caractéristiques fondamentales. Durant ces descriptions de nombreuses références avant apparaîtront inévitablement.

De nombreux tests de type sont réalisés par ces fonctions. Les erreurs suivantes peuvent apparaître : ERNAA, ERNVA, ERNNA et ERNLA dont les libellés par défaut sont :

```
** <fn> : l'argument n'est pas un symbole : <s>
** <fn> : l'argument n'est pas une variable : <s>
** <fn> : l'argument n'est pas un nombre : <s>
** <fn> : l'argument n'est pas une liste : <s>
```

dans lequel le nom le l'argument défectueux <s> est imprimé ainsi que le nom de la fonction <fn> qui a provoqué l'erreur.

3.1 Les fonctions d'évaluation

(EVAL <s>) **[SUBR à 1 argument]**

C'est la fonction principale de l'interprète. EVAL retourne la valeur de l'évaluation de l'argument <s> (voir la description complète de cette fonction dans le chapitre précédent).

```
ex : (EVAL '(1+ 55))                          ->   56
     (EVAL (LIST '+ 8 '(1+ 3) 1))             ->   13
     (EVAL '((CAR '(CDR)) '(A B C)))          ->  (B C)
```

(EVLIS <l>) **[SUBR à 1 argument]**

retourne une liste composée des valeurs des évaluations de tous les éléments de la liste <l>.

EVLIS peut être défini en Lisp de la manière suivante :

```
(DE EVLIS (l)
    (IF (NULL l)
        ()
        (CONS (EVAL (CAR l)) (EVLIS (CDR l)))))

ex : (SETQ L '((1+ 5) (1+ 7) (1+ 9)))
                          ->  ((1+ 5) (1+ 7) (1+ 9))
     (EVLIS L)            ->   (6 8 10)
```

(EPROGN <l>) **[SUBR à 1 argument]**

évalue séquentiellement tous les éléments de la liste <l>. EPROGN retourne la valeur de la dernière évaluation (c'est-à-dire celle du dernier élément de <l>).

EPROGN peut être défini en Lisp de la manière suivante :

```
(DE EPROGN (l)
    (IF (NULL (CDR l))
        (EVAL (CAR l))
        (EVAL (CAR l))
        (EPROGN (CDR l))))

ex : (SETQ L '((PRIN l) (PRIN 2) (PRIN 3)))
     -> ((PRIN l)(PRIN 2)(PRIN 3))
    (EPROGN L)  123  ->  3
```

(PROG1 <s1> ... <sN>) **[FSUBR]**

évalue séquentiellement les différentes expressions <s1> ... <sN>. **PROG1** retourne la valeur de la première évaluation (c'est-à-dire celle de <s1>).

PROG1 peut être défini en Lisp de la manière suivante :

```
(DF PROG1 (first . rest)
    (LET ((result (EVAL first)))
         (EPROGN rest)
         result))
```

ou bien sous la forme d'une EXPR :

```
(DE PROG1 l (CAR l))
```

```
ex : (PROG1 (PRIN 1)(PRIN 2)(PRIN 3))  123  ->  1
```

PROG1 est toujours utilisé pour réaliser des effets de bord. Voici une **MACRO** (EXCH var1 var2) qui réalise l'échange des valeurs des variables **var1 et var2** sans utiliser de mémoire auxiliaire :

```
(DM EXCH (exch var1 var2)
    (LIST 'SETQ var1
          (LIST 'PROG1 var2
                (LIST 'SETQ var2 var1))))))
```

; avec cette MACRO un appel de type :

```
(EXCH var1 var2)
```

; est expansé en :

```
(SETQ var1 (PROG1 var2 (SETQ var2 var1)))
```

(PROGN <s1> ... <sN>) **[FSUBR]**

évalue en séquence les différentes expressions <s1> ... <sN>. PROGN **retourne** la valeur de la dernière évaluation (c'est-à-dire celle de <sN>).

PROGN est la forme FSUBRée de la fonction EPROGN et peut **donc être** décrite sous forme de FEXPR de la manière suivante :

```
(DF PROGN l (EPROGN l))
```

```
; ou bien sous forme d'EXPR :
```

```
(DE PROGN l (CAR (LAST l)))
```

```
ex : (PROGN (PRIN 1)(PRIN 2)(PRIN 3))  123  ->  3
```

(QUOTE <s>) [FSUBR]

retourne la S-expression <s> non-évaluée. Cette fonction est utilisée comme argument des fonctions de type SUBR ou EXPR dont on ne désire pas évaluer les arguments.
Il existe un macro-caractère prédéfini qui facilite cette écriture, le caractère apostrophe *(quote)* ' . Ce caractère placé devant une expression quelconque <s> fabriquera à la lecture la liste (QUOTE <s>).

QUOTE peut être défini en Lisp de la manière suivante :

```
(DF QUOTE (elem) elem)

ex : (QUOTE (1+ 4))    ->  (1+ 4)
     '(A (B C))        ->  (A (B C))
     'A                ->  A
     ''A               ->  (QUOTE A)
     '''A              ->  ''A
     '(QUOTE A B)      ->  (QUOTE A B)
```

3.2 Les fonctions d'application

(LAMBDA <l> <s1> ... <sN>) [FSUBR]
(FLAMBDA <l> <s1> ... <sN>) [FSUBR]
(MLAMBDA <l> <s1> ... <sN>) [FSUBR]

la valeur d'une lambda-expression de type LAMBDA, FLAMBDA ou MLAMBDA est cette lambda-expression elle-même. LAMBDA, FLAMBDA et MLAMBDA ont été rajoutées en tant que fonction pour éviter de *quoter* les lambda-expressions explicites anonymes dans les fonctions d'application.

LAMBDA peut être défini en Lisp de la manière suivante :

```
(DF LAMBDA l (CONS 'LAMBDA l))

ex : (LAMBDA (x) x)                    ->   (LAMBDA (x) x)
     (MAPC (LAMBDA (x) (PRIN x))
           '(A B C)) ABC               ->   NIL
     ((LAMBDA ((x . y) . z) (LIST x y z)) '(1 2) '(3 4))
                          ->   (1 (2) ((3 4)))
```

3.2.1 Les fonctions d'application simples

(APPLY <fn> <l>) **[SUBR à 2 arguments]**

retourne la valeur de l'application de la fonction <fn> à la liste d'arguments <l>. La fonction <fn> peut être une fonction de n'importe quel type SUBR, NSUBR, FSUBR, EXPR, FEXPR ou MACRO.

```
ex : (APPLY '+ '(5 6 7))     ->     18
     (APPLY (LAMBDA (x y) (+ x y)) (LIST (1+ 8) (- 10 3)))
     ->  16
```

(FUNCALL <fn> <s1> ... <sN>) **[SUBR à N arguments]**

est équivalente à la fonction APPLY. Toutefois les arguments de la fonction ne sont pas regroupés sous la forme d'une liste mais sont arguments de la fonction FUNCALL. FUNCALL est donc une autre forme de la fonction APPLY.
L'appel (FUNCALL <fn> <s1> ... <sN>) est équivalent à l'appel (APPLY <fn> (LIST <s1> ... <sN>)).
toutefois il n'y a pas construction d'une liste dans le cas des SUBRs.

FUNCALL peut être défini en Lisp de la manière suivante :

```
(DE FUNCALL l
    (APPLY (CAR l) (CDR l)))
```

```
ex : (FUNCALL (LAMBDA (x y) (CONS x y)) 'A 'B)   ->   (A . B)
     (FUNCALL '+ 5 6 7)                          ->   18
```

(SELF <s1> ... <sN>) **[SUBR à N arguments]**

retourne la valeur de l'application de la dernière LAMBDA-expression utilisée dynamiquement aux différents arguments <s1> ... <sN>. Cette fonction permet de définir des LAMBDA-expressions anonymes récursives et de ce fait est souvent employée à l'intérieur des LET.

SELF ne peut pas se décrire simplement en Lisp.

```
ex : (LET ((l '(A B C D)))
          (IF (NULL (CDR l))
              (CAR l)
              (SELF (CDR l))))
     -> D
```

3.2.2 Les fonctions d'application de type MAP

Ces fonctions permettent d'appliquer une fonction successivement à un ensemble de listes arguments. Les fonction appliquées peuvent avoir un nombre quelconque d'arguments. Dans les descriptions Lisp accompagnant ces fonctions nous utiliserons les fonctions auxiliaires suivantes :

```
(DE ALLCAR (l)
    ; l est une liste de listes
    ; retourne la liste de tous leur CAR
    (IF (NULL l) () (CONS (CAAR l) (ALLCAR (CDR l)))))

(DE ALLCDR (l)
    ; l est une liste de listes
    ; retourne la liste de tous leur CDR
    (IF (NULL l) () (CONS (CDAR l) (ALLCDR (CDR l)))))
```

il est toutefois très clair que le système utilise en fait d'autres techniques extrêmement plus efficaces.

(MAP <fn> <l1> ... <lN>) **[SUBR à N arguments]**

applique successivement la fonction <fn> avec toutes les listes <li> comme arguments puis de nouveau avec tous les CDR de ces listes jusqu'à ce la première de ces listes argument devienne égale à NIL. MAP retourne toujours NIL en valeur.

MAP peut être défini en Lisp de la manière suivante :

```
(DE MAP (f . l)
    (LET ((l l))
         (WHEN (CAR l)
               (APPLY f l)
               (SELF (ALLCDR l)))))

ex : (MAP 'PRINT '(A (B C) D) '(X Y Z))
     (A (B C) D)(X Y Z)
     ((B C) D)(Y Z)
     (D)(Z)
     -> NIL
```

(MAPC <fn> <l1> ... <lN>) **[SUBR à N arguments]**

applique successivement la fonction <fn> avec tous les CAR de toutes les listes <li> comme arguments puis avec tous les CADR puis avec tous les CADDR ... jusqu'à ce que la première liste argument soit égale à NIL. MAPC retourne NIL en valeur.

MAPC peut être défini en Lisp de la manière suivante :

```
(DE MAPC (f . l)
    (LET ((l l))
         (WHEN (CAR l)
               (APPLY f (ALLCAR l))
               (SELF (ALLCDR l)))))

ex: (MAPC 'PRINT '(A (B C) D) '(X Y Z))
    AX
    (B C)Y
    DZ
    -> NIL
```

(MAPLIST <fn> <l1> ... <lN>) **[SUBR à N arguments]**

applique successivement la fonction <fn> à toutes les listes <li> puis à tous leur CDR. MAPLIST est donc identique à la fonction MAP du point de vue application de fonction mais retourne en valeur la liste des valeurs de toutes les applications successives.

MAPLIST peut être défini en Lisp de la manière suivante :

```
(DE MAPLIST (f . l)
    (LET ((l l))
         (WHEN (CAR l)
               (CONS (APPLY f l)
                     (SELF (ALLCDR l))))))

ex : (MAPLIST 'PRINT '(A (B C) D))
     (A (B C) D)
     ((B C) D)
     (D)
     -> ((A (B C) D) ((B C) D) (D))
```

(MAPCAR <fn> <l1> ... <lN>) **[SUBR à N arguments]**

applique successivement la fonction <fn> avec tous les CAR de toutes les listes <li> comme arguments. MAPCAR est donc identique fonctionnellement à la fonction MAPC mais retourne en valeur la liste des valeurs des applications successives.

MAPCAR peut être défini en Lisp de la manière suivante :

```
(DE MAPCAR (f . l)
    (LET ((l l))
         (WHEN (CAR l)
               (CONS (APPLY f (ALLCAR l))
                     (SELF (ALLCDR l))))))

ex: (MAPCAR 'PRINT '(A (B C) D))
    A
    (B C)
    D
```

```
-> (A (B C) D)
(MAPCAR 'CONS '(A B C) '(1 2))   -> ((A . 1) (B . 2) (C))
```

(MAPCON <fn> <l1> ... <lN>) **[SUBR à N arguments]**

applique successivement la fonction <fn> avec toutes les listes <li> comme arguments puis avec tous leur CDR. MAPCON est identique fonctionnellement à la fonction MAP. Toutefois chaque évaluation doit retourner une liste en valeur et MAPCON retourne la liste des valeurs des évaluations qui sont rassemblées au moyen de la fonction NCONC (voir cette fonction).

MAPCON peut être défini en Lisp de la manière suivante :

```
(DE MAPCON (f . l)
    (LET ((l l))
         (WHEN (CAR l)
               (NCONC (APPLY f l)
                      (SELF (ALLCDR l))))))

ex: (MAPCON (LAMBDA (x y) (cons x y))
            '(1 2 3)
            '(4 5 6))
-> ((1 2 3) 4 5 6 (2 3) 5 6 (3) 6)

; que peut bien faire la fonction suivante
; (de Emmanuel Saint-James)

(DE FOOB (l)                    ; prononcez "foube"
    (MAPCON (LAMBDA (s)
                  (UNLESS (MEMQ (CAR s) (CDR s))
                          (LIST (car s))))
            l))
```

(MAPCAN <fn> <l1> ... <lN>) **[SUBR à N arguments]**

applique successivement la fonction <fn> avec tous les CAR de toutes les listes <li> comme arguments. MAPCAN est identique fonctionnellement à la fonction MAPC. Toutefois chaque évaluation doit retourner une liste en valeur et MAPCAN retourne la liste des valeurs des évaluations qui sont rassemblées au moyen de la fonction NCONC (voir cette fonction).

MAPCAN peut être défini en Lisp de la manière suivante :

```
(DE MAPCAN (f . l)
    (LET ((l l))
         (WHEN (CAR l)
               (NCONC (APPLY f (ALLCAR l))
                      (SELF (ALLCDR l))))))

ex: (MAPCAN (LAMBDA (x y) (list (1+ x) (1- y)))
            '(1 2 3)
```

```
                '(1 2 3))
        ->  (2 0 3 1 4 2)
```

(MAPCOBLIST <fn>) **[SUBR à 1 argument]**

applique successivement à la fonction <fn>, qui doit être à 1 argument, tous les éléments de la liste des symboles *(l'OBLIST)*. Chaque évaluation doit retourner une liste et l'ensemble de ces listes est rassemblé au moyen de la fonction NCONC. Si une de ces valeurs n'est pas une liste, elle est ignorée et ne figure donc pas dans la liste résultat. Cette fonction est donc équivalente à :

(MAPCON <fn> (OBLIST))

mais est beaucoup plus efficace en particulier parce qu'elle ne construit pas la liste complète des symboles, liste qui possède typiquement plus de 500 éléments.

ex : fonction qui retourne la liste de toutes les fonctions de type NSUBR du système :

```
(DE FINDNSUBR ()
    (MAPCOBLIST (LAMBDA (sym)
                        (WHEN (EQ (TYPEFN sym) 'NSUBR)
                              (LIST sym)))))

(FINDNSUBR)  ->
(PRINT PRIN FUNCALL LIST NCONC APPEND MAP MAPC MAPCAR
 MAPLIST MAPCON MAPCAN SET + - * MIN MAX ...
```

ex : forme qui retourne tous les symboles contenant un $

```
(MAPCOBLIST (LAMBDA (sym)
                    (WHEN (CHRPOS #/$ sym)
                          (LIST sym))))
-> ($ECODE$ $BCODE$ $UP$ $DOWN$ $LEFT$ $RIGHT$ ...
```

3.2.3 Les fonctions manipulant l'environnement

Ces fonctions vont pouvoir changer l'environnement temporairement. Le terme environnement est pris ici comme l'ensemble des liaisons variable-valeur. Au sortir de ces fonctions l'environnement initial (c'est-à-dire celui avant l'appel de l'une de ces fonctions) est restitué automatiquement.

(LET <l> <s1> ... <sN>) **[MACRO]**

permet d'appeler une fonction anonyme de type EXPR (c'est-à-dire une lambda-expression).
<l> est une liste de la forme :

-() dans le cas où il n'y a pas de variable
-((<var1> <val1>) ... (<varM> <valM>)) dans le cas de variables multiples.

<s1> ... <sN> est un corps de fonction.

La fonction LET va lier dynamiquement et en parallèle toutes les variables <vari> aux valeurs <vali> puis lancer l'exécution du corps de la fonction <s1> ... <sN>. Au sortir du corps de cette fonction les variables seront déliées et retrouveront leurs anciennes valeurs. LET permet ainsi de définir et d'initialiser des variables locales très facilement.

LET est une forme abrégée d'appel de fonctions anonymes de type EXPR.

Ainsi la forme
(LET ((<var> <val>)) <s1> ... <sN>)
correspond à l'appel
((LAMBDA (<var>) <s1> ... <sN>) <val>)

et la forme
(LET ((<var1> <val1>) ... (<varM> <valM>)) <s1> ... <sN>)
correspond à l'appel
((LAMBDA (<var1> ... <varM>) <s1> ... <sN>)
<val1> ... <valM>)

LET peut être défini en Lisp de la manière suivante :

```
(DM LET 1
    (RPLAC 1
           (MCONS LAMBDA (MAPCAR 'CAR (CADR 1)) (CDDR 1))
           (MAPCAR 'CADR (CADR 1))))
```

(LETN <sym> <l> <s1> ... <sN>) [MACRO]

est équivalente à la fonction précédente; toutefois le nom <sym> est temporairement associé à la lambda-expression formée ce qui permet d'utiliser des lambda-expressions dynamiques nommées.

La forme :
(LETN <sym> (<va1><vl1> ... <vaN><vlN>) <s1> ... <sN>)
est équivalente à :
(FLET ((<sym> (<va1> ... <vaN>)
<s1> ... <sN>))
(<sym> <vl1> ... <vlN>))

(ENV <al> <e1> ... <eN>) [FSUBR]

tout d'abord ENV évalue l'argument <al> qui doit retourner une A-liste. Puis, en liant temporairement toutes les variables de cette A-liste aux valeurs correspondantes, ENV évalue séquentiellement les expressions <e1> ... <eN> et retourne la valeur de l'évaluation de <eN>.

(ENVQ <al> <e1> ... <eN>) [FSUBR]

est équivalent à la forme :
(ENV '<al> <e1> ... <eN>)

(BIND <l> <e1> ... <eN>) [FSUBR]

Le_Lisp 80 possède un certain nombre de SUBR à 0 ou 1 argument qui permettent d'accéder et/ou de modifier les variables internes du système et de SUBR à 1 ou 2 arguments qui permettent d'accéder et/ou de modifier des vecteurs système ou bien les propriétés naturelles des symboles. Voici la liste de ces fonctions : PROMPT, ICASE, SPRINT, OBASE, LMARGIN, RMARGIN, OUTPOS, INPOS, INMAX, PRINTLENGTH, PRINTLINE, PRINTLEVEL, MEMORYB, TYPECH, TYPECN, OUTBUF, INBUF, PLIST, CVAL, VALFN, TYPEFN, PTYPE.
BIND va permettre de lier dynamiquement ces variables internes. <l> est une liste d'appel de ce type de SUBR. BIND va effectuer les liaisons puis évaluer le corps <e1> ... <eN>. Au retour de cette évaluation, les anciennes valeurs de ces variables internes seront restaurées.

```
ex : (BIND ((LMARGIN 0)     ; avec une marge gauche a 0
           (RMARGIN 72)     ; une marge droite a 72
           (OBASE 10))      ; et une base de sortie de 10,
       .....                   ... faire ...
     )                      ; LMARGIN, RMARGIN et OBASE
                          ; retrouvent leurs anciennes valeurs.
```

3.3 Les fonctions de définition de fonctions

Ces fonctions vont permettre de définir de nouvelles fonctions. Elles testent la validité de leurs arguments :
- les noms des fonctions doivent être des symboles
- toutes les variables de la liste des paramètres formels doivent également être des symboles.

Si ce n'est pas le cas, une erreur de type apparait.

Il existe 2 types de définition de fonction : les définitions statiques et les définitions dynamiques (voir le chapitre précédent sur le fonctionnement de l'interprète).

3.3.1 Les définitions des fonctions statiques

Toutes ces fonctions vont changer de façon permanente les fonctions associées aux symboles.

(DE <sym> <lvar> <s1> ... <sN>) **[FSUBR]**

permet de définir statiquement de nouvelles EXPR. <sym> est le nom du symbole auquel sera rattaché une lambda-expression de type :

(LAMBDA <lvar> <s1> ... <sN>)

DE (prononcez *dé* comme en latin) retourne le nom de la fonction <sym> en valeur.

```
ex : (DE FOO (l) (IF (NULL (CDR l)) l (FOO (CDR l)))  ->  FOO
     (GETDEF 'FOO))
     ->  (DE FOO (l) (IF (NULL (CDR l)) l (FOO (CDR l))))
     (TYPEFN 'FOO)    ->  EXPR
     (FOO '(A B C))   ->  (C)
```

(DF <sym> <lvar> <s1> ... <sN>) **[FSUBR]**

permet de définir statiquement de nouvelles FEXPR. <sym> est le nom du symbole auquel sera rattaché une lambda expression de type :

(FLAMBDA <lvar> <s1> ... <sN>)

DF retourne en valeur le nom <sym> de la fonction définie.

```
ex : (DF INCR (var val)
           (SET var (IF val
                        (+ (EVAL var) (EVAL val))
```

```
                                    (1+ (EVAL var)))))  -> INCR
(TYPEFN 'INCR)                                  -> FEXPR
(SETQ nb 7)                                     -> 7
(INCR nb)                                       -> 8
nb                                              -> 8
```

(DM <sym> <lvar> <s1> ... <sN>) **[FSUBR]**

permet de définir statiquement de nouvelles MACRO. <sym> est le nom du symbole auquel sera rattaché une lambda expression de type :

(MLAMBDA <lvar> <s1> ... <sN>)

DM retourne en valeur le nom <sym> de la fonction définie.

```
ex : (DM DECR (decr var val)
         (LIST 'SETQ var
               (IF val
                   (LIST '+ var val)
                   (LIST '1- (CADR 1)))))
     ->  DECR
(SETQ N 10)  ->  10
(DECR N)     ->  9
(DECR N 2)   ->  7
```

3.3.2 L'accès aux définitions des fonctions

(SYNONYM <sym1> <sym2>) [SUBR à 2 arguments]

permet de donner au symbole <sym1> le type et la valeur de la fonction associée au symbole <sym2>.

SYNONYM peut être défini en Lisp de la manière suivante :

```
(DE SYNONYM (at1 at2)
    (SET at1 at2)
    (PLIST at1 (PLIST at2))
    (TYPEFN at1 (TYPEFN at2))
    (VALFN at1 (VALFN at2))
    (PTYPE at1 (PTYPE at2))
    at1)

ex :  (SYNONYM 'KONS 'CONS)   ->  KONS
      (KONS 'A 'B)            ->  (A . B)
```

(TYPEFN <sym> <s>) **[SUBR à 1 ou 2 arguments]**

si le 2ème argument <s> est fourni il devient le nouveau type de la fonction associée au symbole <sym>. Dans tous les cas TYPEFN retourne le type courant de la fonction associée au symbole <sym> ou NIL si le symbole ne possède pas de définition de fonction. Le type d'une fonction peut être l'un des atomes suivant : EXPR, FEXPR, MACRO, 0SUBR, 1SUBR, 2SUBR, 3SUBR, SUBR1V, SUBR2V, NSUBR ou FSUBR.

(VALFN <sym> <s>) **[SUBR à 1 ou 2 arguments]**

si le 2ème argument <s> est fourni il devient la nouvelle valeur du symbole <sym> considéré comme une fonction. Dans tous les cas VALFN retourne la liste des variables et le corps de la fonction associée au symbole <sym>.

(GETDEF <sym>) **[SUBR à 1 argument]**

permet de retourner la définition de la fonction associée au symbole <sym> sous la forme de sa définition (c'est-à-dire sous la forme de l'appel d'une des fonctions DE, DF, DM ou DMC).

```
ex :  (DE BAR (n) (+ n n))  ->  BAR
      (TYPEFN 'BAR)         ->  EXPR
      (VALFN  'BAR)         ->  ((n) (+ n n))
      (GETDEF 'BAR)         ->  (DE BAR (n) (+ n n))
```

(FINDFN <s>) **[SUBR à 1 argument]**

recherche dans toute l'OBLIST le symbole dont la valeur en tant que fonction est égale à <s>. FINDFN retourne ce symbole si il existe et NIL dans le cas contraire.

FINDFN peut être défini en Lisp de la manière suivante :

```
(DE FINDFN (s)
    (TAG trouve
         (MAPCOBLIST
              (LAMBDA (sym)
                      (WHEN (EQ (VALFN sym) s)
                            (EXIT trouve sym))))))
```

```
ex : (DE FOO (x) (+ x 2))    ->  FOO
     (VALFN 'FOO)            ->  ((x) (+ x 2))
     (FINDFN (VALFN 'FOO))   ->  FOO
```

(REMFN <sym>) **[SUBR à 1 argument]**

détruit la fonction associée au symbole <sym>. Toute utilisation postérieure du symbole <sym> en tant que fonction déclenchera l'erreur ERUDF. REMFN retourne <sym> en valeur.

```
ex : (DE BAR (x y) (+ (1- x) (1- y)))   ->   BAR
     (BAR 5 6)                          ->   9
     (REMFN 'BAR)                       ->   BAR
     (BAR 1 2)
      ** EVAL : fonction indefinie : BAR
```

3.3.3 Les définitions des fonctions dynamiques

(FLET <l> <s1> ... <sN>) **[FSUBR]**

permet de définir des fonctions dynamiquement (d'une manière analogue à la fonction LET pour les variables).

<l> est une liste de liste. Chaque sous-liste à la forme :
(<sym> <l> <e1> ... <eN>)
<s1> ... <sN> est un corps de fonction.

FLET va lier, le temps de l'évaluation de <s1> ... <sN>, une nouvelle fonction à chacun des symboles <sym>. Ces fonctions, qui sont toujours de type EXPR, vont possèder une liste d'arguments <l> et un corps <e1> ... <eN>. La valeur retournée par FLET est la valeur de la dernière évaluation du corps du FLET c'est-à-dire celle de <sN>. Au sortir du FLET, les symboles <sym> reprendront leur ancienne définition (s'ils en avaient une).

FLET est utilisé en général pour redéfinir temporairement les fonctions systèmes comme EOL, BOL, GCALARM ...

```
ex : (FLET ((CAR (x) (CDR x))) (CAR '(A B C)))  ->  (B C)
     (CAR '(A B C))                             ->  A
     (FLET ((CAR (x) (CDR x))
            (CDR (x) (CDDR x)))
           (CDR (CAR '(A B C D E))))            ->  (E)
```

3.4 Les fonctions de contrôle

Toutes les fonctions de cette section vont permettre de rompre le déroulement séquentiel des évaluations. C'est la raison pour laquelle elles sont toutes de type FSUBR c'est-à-dire que les arguments ne seront pas évalués par EVAL mais par les fonctions elles-mêmes et cela sélectivement.

La fonction de contrôle la plus simple est la fonction conditionnelle IF. Cette fonction va être utilisée avec la fonction WHILE pour décrire en Lisp toutes les autres fonctions de contrôle sous forme de FEXPR. Rappelons qu'à l'appel d'une fonction de type FEXPR la liste des arguments non évalués (c'est-à-dire le CDR de l'appel lui-même) est liée à l'arbre des variables.

3.4.1 Les fonctions de contrôle de base

(IF <s1> <s2> <s3> ... <sN>) **[FSUBR]**

si la valeur de l'évaluation de <s1> est différente de NIL, IF retourne la valeur de l'évaluation de l'expression <s2>, sinon IF évalue en séquence les différentes expressions <s3> ... <sN> et retourne la valeur de la dernière évaluation <sN>. IF permet de construire une structure de contrôle de type :

si <s1> alors <s2> sinon <s3> ... <sN>

```
ex : (IF T 1 2 3)     ->  1
     (IF NIL 1 2 3)   ->  3

     ; voici la fonction d'ACKERMANN decrite avec des IF

     (DE ACK (x y)
         (IF (= x 0)
             (1+ y)
             (ACK (1- x)
                  (IF (= y 0)
                      1
                      (ACK x (1- y)))))))
```

(IFN <s1> <s2> <s3> ... <sN>) **[FSUBR]**

si la valeur de l'évaluation de <s1> est égale à NIL, IFN retourne la valeur de l'évaluation de l'expression <s2>, sinon IFN évalue en séquence les différentes expressions <s3> ... <sN> et retourne la valeur de la dernière évaluation <sN>. IFN permet de construire une structure de contrôle de type :

si non <s1> alors <s2> sinon <s3> ... <sN>

IFN est donc équivalent à (IF (NULL <s1>) <s2> <s3> ... <sN>).

IFN peut être défini en Lisp de la manière suivante :

```
(DF IFN (test then . else)
    (IF (NULL (EVAL test))
        (EVAL then)
        (EPROGN else)))

ex : (IFN T 1 2 3)    -> 3
     (IFN NIL 1 2 3)  -> 1
```

(WHEN <s1> <s2> ... <sN>) **[FSUBR]**

si la valeur de l'évaluation de <s1> est différente de NIL, WHEN évalue en séquence les différentes expressions <s2> ... <sN> et retourne la valeur de la dernière évaluation <sN>. Si la valeur de <s1> est égale à NIL, WHEN retourne NIL. WHEN permet de construire une structure de contrôle de type :

si <s1> alors <s2> ... <sN> sinon NIL

WHEN peut être défini en Lisp de la manière suivante :

```
(DF WHEN (test . body)
    (IF (EVAL test)
        (EPROGN body)
        NIL))

ex : (WHEN T 1 2 3)    -> 3
     (WHEN NIL 1 2 3)  -> NIL
```

(UNLESS <s1> <s2> ... <sN>) **[FSUBR]**

si la valeur de l'évaluation de <s1> est égale à NIL, UNLESS évalue en séquence les différentes expressions <s2> ... <sN> et retourne la valeur de la dernière évaluation <sN>. Si la valeur de <s1> est différente de NIL, UNLESS retourne NIL. UNLESS permet de construire une structure de contrôle de type :

si <s1> alors NIL sinon <s2> ... <sN>

UNLESS peut être défini en Lisp de la manière suivante :

```
(DF UNLESS (test . body)
    (IF (EVAL test)
        NIL
        (EPROGN body)))
```

```
ex : (UNLESS T 1 2 3)    ->  NIL
     (UNLESS NIL 1 2 3)  ->  3
```

(OR <s1> ... <sN>) **[FSUBR]**

évalue successivement les différentes expressions <s1> ... <sN> jusqu'à ce que l'une de ces évaluations ait une valeur différente de NIL. OR retourne cette valeur.

OR peut être défini en Lisp de la manière suivante :

```
(DF OR 1
    (LETN OREVAL ((l 1))
          (IF (NULL (CDR l))
              (EVAL (CAR l))
              (LET ((resul (EVAL (CAR l))))
                   (IF resul
                       resul
                       (OREVAL (CDR l)))))))
```

```
ex : (OR NIL NIL 2 3)   ->  2
```

(AND <s1> ... <sN>) **[FSUBR]**

évalue successivement les différentes expressions <s1> ... <sN> jusqu'à ce que la valeur d'une évaluation soit égale à NIL, à ce moment AND retourne NIL sinon AND retourne la valeur de la dernière évaluation <sN>. Si aucune expression n'est fournie, cette fonction retourne T. AND permet de construire une structure de contrôle de type :

si <s1> alors si <s2> alors ... <sN>

AND peut être défini en Lisp de la manière suivante :

```
(DF AND l
    (IF (NULL l)
        T
        (LETN ANDEVAL ((l l))
              (IF (NULL (CDR l))
                  (EVAL (CAR l))
                  (IF (EVAL (CAR l))
                      (ANDEVAL (CDR l))
                      NIL)))))
```

```
ex : (AND)          ->  T
     (AND NIL)      ->  NIL
```

```
(AND 1 2 3 4)    ->  4
(AND 1 2 () 4)   ->  NIL
```

(COND <l1> ... <lN>) [FSUBR]

est la fonction conditionnelle la plus ancienne, la plus compatible et la plus générale de Lisp. Les différents arguments <l1> ... <lN> sont des listes appelées clauses qui ont la structure suivante :

(<ss> <s1> ... <sN>)

COND va sélectionner une seule de ces clauses en les parcourant séquentiellement : la première dont la valeur de l'évaluation de son premier élément <ss> est différente de NIL. COND évalue alors en séquence les différentes expressions <s1> ... <sN> de la clause sélectionnée et retourne la valeur de la dernière évaluation <sN>. Si la clause sélectionnée n'a qu'un élément <ss>, COND retourne la valeur de l'évaluation de <ss> (c'est-à-dire la valeur qui a déclenché la sélection de cette clause). COND permet de construire des structures de contrôle de type :

si ... alors ... sinon si ... alors ... ...

Si aucune clause n'est sélectionnée, COND retourne NIL. Pour forcer la sélection de la dernière clause, il est d'usage d'utiliser comme sélecteur l'atome T dont la valeur est toujours différente de NIL (voir la section sur les prédicats).

COND peut être défini en Lisp de la manière suivante :

```
(DF COND 1
    (LETN CONDEVAL ((l 1))
          (WHEN l
                (IF (OR (ATOM l) (ATOM (CAR l)))
                    (SYSERROR 'COND 'ERNLA l)
                    (LET ((select (EVAL (CAAR l))))
                         (IF select
                             (IF (CDAR l)
                                 (EPROGN (CDAR l))
                                 select)
                             (CONDEVAL (CDR l)))))))))

; donc :          est équivalent à :
      (COND                    (COND
        (p1 e11 e12 e13)         (p1 (PROGN e11 e12 e13))
        (p2 e21 e22)             (p2 (PROGN e21 e22))
        (p3)                     ((LET ((aux p3)) aux))
        (p4 e41))                (p4 e41))

ex : (COND (NIL 1 2) (T 3 4 5))   ->  5

     (COND ((ATOM X) 'ATOM)
           ((LISTP (CAR X))  'LIST)
           ((NULL (CDR X)) 'NULL)
           (T 'What???))
```

(SELECTQ <s> <l1> ... <lN>) [FSUBR]

comme pour la fonction COND, SELECTQ va sélecter une des clauses <l1> ... <lN>. Le sélecteur de ces clauses est la valeur de l'évaluation de <s>, la sélection s'effectue par comparaison du sélecteur avec :
- le CAR (non évalué) de la clause si celui-ci est un symbole (en utilisant le prédicat EQ).
- les différents éléments du sélecteur si celui-ci est une liste (en utilisant la fonction de recherche MEMBER). Cette dernière possibilité permet donc de sélecter des objets de n'importe quel type et ce en nombre quelconque.

Dès qu'une clause est sélectée, SELECTQ évalue en séquence le reste de la clause et retourne la valeur de la dernière évaluation.
Si aucune des clauses <l1> ... <lN> n'est sélectée, SELECTQ retourne NIL.
SELECTQ permet donc de construire des aiguillages sur valeurs constantes.
Avec la même syntaxe que pour la fonction COND, il est possible de forcer la sélection de la dernière clause en utilisant l'atome T en position sélecteur.

SELECTQ peut être défini en Lisp de la manière suivante :

```
(DF SELECTQ (sel . cl)
    (LET ((sel (EVAL sel)))
         ; sel = le selecteur evalue
         (LET ((cl cl))
              ; cl = les clauses non evaluees
              (COND
                  ((NULL cl)
                     ; plus de clauses
                     ())
                  ((OR (ATOM cl) (ATOM (CAR cl))
                     ; une mauvais clause
                     (SYSERROR 'SELECTQ 'ERNLA cl))
                  ((EQ (CAAR cl) T)
                     ; selection de la clause toujours vraie
                     (EPROGN (CDAR cl)))
                  ((AND (SYMBOLP (CAAR cl))
                        (EQ sel (CAAR  cl)))
                     ; selection simple
                     (EPROGN (CDAR cl)))
                  ((AND (LISTP (CAAR cl))
                        (MEMBER sel (CAAR  cl)))
                     ; selection generale
                     (EPROGN (CDAR cl)))
                  (T (SELF (CDR cl)))))))))

ex : (SELECTQ 'ROUGE
        (VERT 'ESPOIR)
        (ROUGE 'OK)
        (T 'NON))
     -> OK
```

```
(SELECTQ 'ROUGE
   ((BLEU VERT ROUGE) 'COULEUR)
   ((ROSE IRIS) 'FLEUR)
   (T 'SAIS-PAS))
-> COULEUR
```

(WHILE <s> <s1> ... <sN>) **[FSUBR]**

tant que la valeur de l'évaluation de <s> est différente de NIL, WHILE va évaluer en séquence les différentes expressions <s1> ... <sN>. WHILE retourne toujours NIL en valeur (qui est la dernière évaluation de <s> qui fait sortir de la boucle WHILE). Cette fonction permet de construire des boucles conditionnelles d'une manière fort commode, ainsi que des boucles infinies en utilisant la forme : (WHILE T ...) car l'atome T possède toujours une valeur différente de NIL. Enfin signalons que le test a lieu avant d'entrer dans la boucle.

WHILE peut être défini en Lisp de la manière suivante :

```
(DF WHILE (test . l)
    (LET ()
         (OR (EVAL test))
             (PROGN (EPROGN l)
                    (SELF)))))
```

```
ex : (SETQ S '(A B C D))                    -> (A B C D)
     (WHILE S (PRIN (NEXTL S))) ABCD        -> NIL
```

(UNTIL <s> <s1> ... <sN>) **[FSUBR]**

tant que la valeur de l'évaluation de <s> est égale à NIL, UNTIL va évaluer en séquence les différentes expressions <s1> ... <sN>. UNTIL retourne la valeur de la 1ère évaluation de <s> différente de NIL (celle qui fait sortir de la boucle UNTIL). Comme la fonction précédente, UNTIL permet de construire des boucles conditionnelles d'une manière fort commode, ainsi que des boucles infinies en utilisant la forme : (UNTIL () ...). Enfin signalons que le test a lieu avant d'entrer dans la boucle.

UNTIL peut être défini en Lisp de la manière suivante :

```
(DF UNTIL (test . l)
    (LET ()
         (UNLESS (EVAL test)
                 (EPROGN l)
                 (SELF))))
```

```
ex : (SETQ S '(A B C D))                        -> (A B C D)
     (UNTIL (NULL S) (PRIN (NEXTL S))) ABCD     -> T
```

(REPEAT <n> <s1> ... <sN>) **[FSUBR]**

évalue l'expression <n>. Cette valeur doit être un nombre <n>. REPEAT évalue alors <n> fois les différentes expressions <s1> ... <sN>. Si l'évaluation de <n> n'est pas un nombre strictement positif, la boucle n'est pas exécutée. Dans tous les cas REPEAT retourne la valeur T.

REPEAT peut être défini en Lisp de la manière suivante :

```
(DF REPEAT (n . l)
    (LET ((n (OR (NUMBERP (EVAL n))
                 (SYSERROR 'REPEAT 'ERNNA n))))
         (WHEN (>= n 0)
               (EPROGN l)
               (SELF (1- n))))
      T)

ex : (REPEAT 10 (PRIN '*)) **********   ->  T
     (REPEAT 0 (PRIN '*))  ->  T
```

3.4.2 Les fonctions de contrôle non locales

Les fonctions de contrôle qui vont suivre sont très puissantes et très rapides à l'interprétation. Leur usage est nécessaire pour tout ce qui est structure de contrôle un peu sophistiquée, et est vivement recommandé.
On appelle *échappement* un point de retour défini dynamiquement. Le_Lisp associe à ces échappements des symboles. Ces associations ne mettent en jeu ni la valeur de ces symboles, ni les fonctions associées à ces symboles.

(TAG <sym> <s1> ... <sN>) **[FSUBR]**

permet de définir un échappement de nom <sym>. Ce nom n'est pas évalué. Puis les différentes expressions <s1> ... <sN> sont évaluées. Si aucun appel de la fonction EXIT n'est réalisé durant ces évaluations, TAG retourne la valeur de la dernière évaluation (c'est-à-dire celle de <sN>). En revanche si au cours de ces évaluations un appel du type (EXIT <sym> <e1> ... <eN>) est rencontré, les évaluations à l'intérieur du TAG sont stoppées, les expressions <e1> ... <eN> sont évaluées et <eN> devient la valeur de retour de la fonction TAG.

```
ex : (DE PRESENT (l e)
         (TAG trouve (PRESENTL l)))

     (DE PRESENTL (l)
         (COND ((NULL l) ())
               ((EQ l e) (EXIT trouve l))
```

```
                ((CONSP 1) (PRESENTL (CAR 1)) (PRESENTL (CDR 1)))
                (T ())))))

(PRESENT '(1 (2 . 3) 4) 3)     -> 3
(PRESENT '(1 (2 . 3) 4) 5)     -> NIL
```

(EXIT <sym> <s1> ... <sN>) **[FSUBR]**

permet de sortir de l'échappement de nom <sym> (c'est-à-dire retourner au dernier (TAG <sym> ...) défini dynamiquement) après avoir évalué les différentes expressions <s1> ... <sN>. Si <sym> n'est pas le nom d'un échappement, EXIT déclenche l'erreur ERUDT dont le libellé est :

```
** <fn> : echappement indefini : <sym>
```

dans lequel <fn> est la fonction appelée (EXIT ou EVEXIT) et <sym> est le nom de l'échappement.

(EVEXIT <s> <s1> ... <sN>) **[FSUBR]**

équivalent à EXIT mais le nom de l'échappement est évalué (calculé). C'est, avec LOCK, une des fonctions primitives qui permettent de décrire toutes les autres fonctions sur les échappements.

(UNTILEXIT <sym> <e1> ... <eN>) **[FSUBR]**

permet de réaliser une structure de boucle contrôlée par un échappement de nom <sym>. Les expressions <e1> ... <eN> vont être répétées indéfiniment jusqu'à ce qu'un appel explicite de l'échappement <sym> (ou d'un échappement plus global). UNTILEXIT retourne la valeur ramenée par cet échappement.

```
(UNTILEXIT <sym> <s1> ... <sN>)
```

; correspond donc à l'appel :

```
(TAG <sym> (WHILE T <s1> ... <sN>))
```

(LOCK <fn> <s1> ... <sN>) **[FSUBR]**

en premier lieu LOCK évalue l'argument <fn> qui doit retourner en valeur une fonction à 2 arguments. Cette fonction se nomme la *fonction de verrouillage* Puis LOCK évalue comme un PROGN les différentes expressions <s1> ... <sN>. Si au cours de ces évaluations un échappement se produit, la fonction de verrouillage est automatiquement appelée avec 2 arguments : le premier est le nom de l'échappement en cours et le second la valeur que retourne cet échappement. La valeur retournée par cette fonction de verrouillage (s'il en en retourne une) devient la valeur de la fonction LOCK. Si l'évaluation du corps du LOCK se termine normalement, la fonction de verrouillage est également appelée avec un nom d'échappement égal à NIL et la valeur retournée par cette fonctions de verrouillage devient la valeur de LOCK.

Voici une fonction de verrouillage qui est ineffective car elle relance un échappement de même nom avec la même valeur.

```
(LAMBDA (tag val) (IF tag (EVEXIT tag val) val))
```

Voici une fonction de verrouillage qui arrête tous les échappements en provquant une erreur :

```
(LAMBDA (tag val)
        (IF tag (SYSERROR 'LOCK 'ERUDT [tag val]) val))
```

(PROTECT <s1> <s2> ... <sN>) [FSUBR]

évalue les différentes expressions <s1> ... <sN> et retourne la valeur de la 1ère évaluation (c'est-à-dire celle de <s1>). Son comportement dans le cas le plus simple est donc équivalent à la fonction PROG1. Toutefois si au cours de l'évaluation de <s1> un échappement se produit, les expressions <s2> ... <sN> seront quand même évaluées durant le processus de restauration du contexte de sortie de l'échappement. Cette fonction, indispensable pour l'écriture de sous-systèmes Lisp, permet par exemple de restaurer automatiquement des environnements complexes en cas de sorties extraordinaires ou d'erreurs.

PROTECT peut être défini en Lisp de la manière suivante :

```
(DF PROTECT (el . l)
    (LOCK (LAMBDA (tag val)
                  (EPROGN l)
                  (IF tag
                      (EVEXIT tag val)
                      val))
          (EVAL el)))
```

3.5 Les prédicats de base

En Lisp la valeur booléene *fausse* est assimilée à la valeur NIL et la valeur booléenne *vraie* est assimilée à toute valeur différente de NIL (ce qui laisse un très grand choix de représentation ...).
Certains prédicats devant retourner une valeur booléenne *vraie* utiliseront le symbole spécial T (initiale du TRUE anglais) qui par définition possède une valeur différente de NIL (la valeur du symbole T est le symbole T lui-même).
La règle d'utilisation de cette valeur est la suivante : tout prédicat qui est faux pour la valeur NIL retourne l'argument si le prédicat est vérifié et NIL dans le cas contraire; tout prédicat qui est vrai pour la valeur NIL retourne T si le prédicat est vérifié et NIL dans le cas contraire.

Cette section ne décrit que les prédicats de base. Les tests de l'arithmétique entière et mixte sont décrits dans les sections suivantes.

(NULL <s>) **[SUBR à 1 argument]**

teste si <s> est égal à NIL. NULL retourne T si le test est vérifié et NIL dans le cas contraire.

NULL peut être défini en Lisp de la manière suivante :

```
(DE NULL (s)
    (EQ s NIL))
```

```
ex : (NULL NIL)   ->  T
     (NULL T)     ->  NIL
```

(ATOM <s>) **[SUBR à 1 argument]**

teste si <s> est un atome, c'est-à-dire un symbole ou un nombre. ATOM retourne T si ce test est vérifié, NIL dans le cas contraire.

```
ex : (ATOM ())        ->  T
     (ATOM 'ARGH)     ->  T
     (ATOM 42)        ->  T
     (ATOM '(A B))    ->  NIL
```

(SYMBOLP <s>) **[SUBR à 1 argument]**

teste si <s> est un symbole. SYMBOLP retourne T si le test est vérifié et NIL dans le cas contraire.

```
ex : (SYMBOLP ())        ->  T
     (SYMBOLP 'ARGH)     ->  T
     (SYMBOLP 44)        ->  NIL
     (SYMBOLP '(A B))    ->  NIL
```

(CONSTANTP <s>) [SUBR à 1 argument]

teste si <s> est une constante, c'est-à-dire un objet qui évalué fournit le même objet. Sont des constantes tous les nombres et les symboles NIL et T. CONSTANTP retourne T si le test est vérifié et NIL dans le cas contraire.

```
ex : (CONSTANTP 123)      -> T
     (CONSTANTP NIL)      -> T
     (CONSTANTP 'ARGH)    -> NIL
     (CONSTANTP '(A B))   -> NIL
```

(NUMBERP <s>) [SUBR à 1 argument]

teste si <s> est un nombre. NUMBERP retourne le nombre <s> si le test est vérifié et NIL dans le cas contraire. Les tests des différents types de nombres sont décrits dans la section sur les prédicats numériques.

```
ex : (NUMBERP ())         -> NIL
     (NUMBERP 'ARGH)      -> NIL
     (NUMBERP 44)         -> 44
     (NUMBERP '(A B))     -> NIL
```

(CONSP <s>) [SUBR à 1 argument]

teste si <s> est une liste non vide. CONSP retourne <s> si le test est vérifié et NIL dans le cas contraire. CONSP est le prédicat inverse de ATOM.

```
ex : (CONSP ())           -> NIL
     (CONSP 'ARGH)        -> NIL
     (CONSP 44)           -> NIL
     (CONSP '(A B))       -> (A B)
```

(LISTP <s>) [SUBR à 1 argument]

teste si <s> est une liste qui peut être vide (c'est-à-dire égale au symbole NIL). LISTP retourne le symbole T si le test est vérifié et NIL dans le cas contraire. La représentation de la liste vide sous la forme du symbole NIL pose le problème du statut de ce symbole : NIL est à la fois un symbole et la représentation de la liste vide. Le_Lisp, comme la plupart des post-maclisp Lisps, résout cette ambiguité en introduisant deux fonctions testant les listes, CONSP et LISTP.

```
ex : (LISTP NIL)          -> T
     (LISTP 'ARGH)        -> NIL
     (LISTP 44)           -> NIL
     (LISTP '(A B))       -> T
```

(NLISTP <s>) [SUBR à 1 argument]

teste si <s> n'est pas une liste qui peut être vide. LISTP retourne l'argument si le test est vérifié et NIL dans le cas contraire. Cette fonction est donc l'inverse de la fonction précédente (à la valeur retournée près).

```
ex : (NLISTP NIL)       ->  NIL
     (NLISTP 'ARGH)     ->  ARGH
     (NLISTP  44)       ->  44
     (NLISTP '(A B))    ->  NIL
```

(EQ <s1> <s2>) [SUBR à 2 arguments]

sert à tester 2 symboles. EQ retourne T si les 2 symboles <s1> et <s2> sont égaux et NIL s'ils ne le sont pas. Dans le cas où les arguments ne seraient pas des symboles, la fonction EQ va tester l'égalité des représentations internes des arguments <s1> et <s2> (EQ teste si les deux arguments ont la même adresse physique).
La représentation interne des nombres variant en fonction des différentes implantations du système, il est recommandé d'utiliser la fonction = (qui est décrite avec les prédicats numériques) pour tester l'égalité des valeurs numériques.

```
ex : (EQ 'A (CAR '(A)))      ->  T
     (EQ (1+ 119) 120)       ->  T ou NIL
                                 (en fonction des implantations)
     (EQ '(A B) '(A B))      ->  NIL
     (SETQ L '(x y))         ->  (x y)
     (EQ L L)                ->  T
```

(NEQ <s1> <s2>) [SUBR à 2 arguments]

est équivalent à (NOT (EQ <s1> <s2>)).

NEQ peut être défini en Lisp de la manière suivante :

```
(DE NEQ (s1 s2)
    (IF (EQ s1 s2) NIL T))
```

(EQUAL <s1> <s2>) [SUBR à 2 arguments]

est la fonction de comparaison la plus générale et doit être utilisée dès que le type des objets à comparer n'est pas précisément connu. Si <s1> et <s2> sont des symboles, EQUAL est identique à la fonction EQ. Si <s1> et <s2> sont des nombres, EQUAL est identique à la fonction =, sinon si <s1> et <s2> sont des listes, EQUAL teste si elles possèdent la même structure (c'est-à-dire si les listes possèdent les mêmes éléments). Si le test est verifié, EQUAL retourne T dans le cas contraire EQUAL retourne NIL.

EQUAL peut être défini en Lisp de la manière suivante :

```
(DE EQUAL (l1 l2)
    (COND ((SYMBOLP l1)
```

```
                (IF (SYMBOLP 12) (EQ 11 12) ()))
            ((NUMBERP 11)
                (IF (NUMBERP 12)
                    (IF (= 11 12) T ())))
            ((ATOM 12) ())
            ((AND (EQUAL (CAR 11) (CAR 12))
                  (EQUAL (CDR 11) (CDR 12))))

ex : (EQUAL 1214 (1+ 1213))                    -> T
     (EQUAL '(A (B . C) D) '(A (B . C) D))     -> T
     (EQUAL '(A B C) '(A B C D))               -> NIL
```

(NEQUAL <s1> <s2>) **[SUBR à 2 arguments]**

est équivalent à (NOT (EQUAL <s1> <s2>)).

NEQUAL peut être défini en Lisp de la manière suivante :

```
(DE NEQUAL (s1 s2)
    (IF (EQUAL s1 s2) NIL T))
```

3.6 Les fonctions sur les listes

3.6.1 Les fonctions de recherche dans les listes

Toutes ces fonctions acceptent comme argument <l> une liste ou bien l'atome NIL. Dans tous les autres cas une erreur de type (ERNLA) est déclenchée.

(CAR <l>) **[SUBR à 1 argument]**

si <l> est une liste, retourne son premier élément. Si <l> égal NIL, CAR retourne NIL. Le CAR d'un symbole autre que NIL ou d'un nombre est indéterminé et provoque de ce fait une erreur.

```
ex : (CAR '(A B C))     -> A
     (SETQ X '(U P))    -> (U P)
     (CAR X)            -> U
     (CAR 'X)           ->
     ** CAR : l'argument n'est pas une liste : X
```

(CDR <l>) [SUBR à 1 argument]

si <l> est une liste, retourne cette liste sans son premier élément. Le CDR de NIL est NIL par définition. Le CDR d'un symbole autre que NIL ou d'un nombre est indéterminé et provoque également une erreur.

```
ex : (CDR '(A B C))        ->  (B C)
     (CDR '(X Y))          -> (Y)
     (CDR '(X . Y))        ->  Y
     (CDR NIL)             ->  NIL
```

(C...R <l>) [SUBR à 1 argument]

les 12 combinaisons de CAR et de CDR imbriqués (jusqu'à 3 niveaux) sont disponibles.

(CADR <l>) est équivalent à (CAR (CDR <l>))
(CDAAR <l>) est équivalent à (CDR (CAR (CAR <l>)))

(MEMQ <sym> <l>) [SUBR à 2 arguments]

si le symbole <sym> est un élément de la liste <l>, retourne la partie de la liste <l> commençant au symbole <sym>, sinon retourne NIL. Cette fonction utilise le prédicat EQ pour tester la présence du symbole <sym> dans la liste <l>. On utilise souvent cette fonction comme prédicat pour tester la présence d'un élément de liste dans une liste donnée.

MEMQ peut être défini en Lisp de la manière suivante :

```
(DE MEMQ (at l)
    (IF (OR (ATOM l) (EQ at (CAR l)))
        l
        (MEMQ at (CDR l))))

ex : (MEMQ 'C '(A B C D E))    ->  (C D E)
     (MEMQ 'Z '(A B C D E))    ->  NIL
```

(MEMBER <s> <l>) [SUBR à 2 arguments]

si l'expression <s> est un élément de la liste <l>, retourne la partie de la liste <l> commençant à l'élément <s>, sinon retourne NIL. Cette fonction utilise le prédicat EQUAL et permet donc de rechercher un élément de n'importe quel type dans une liste.

MEMBER peut être défini en Lisp de la manière suivante :

```
(DE MEMBER (s l)
    (IF (OR (ATOM l) (EQUAL s (CAR l)))
        l
        (MEMBER s (CDR l))))
```

```
ex : (MEMBER 'C '(A B C D E))        ->  (C D E)
     (MEMBER 'Z '(A B C D E))        ->  NIL
     (MEMBER '(A B) '(A (A B) C))    ->  ((A B) C)
```

(NTHCDR <n> <l>) **[SUBR à 2 arguments]**

retourne la partie de la liste <l> commençant à son <n>ième CDR. Si <l> n'est pas une liste ou si (LENGTH <l>) est plus petit que <n>, NTH retourne NIL. Si <n> <= 0, NTHCDR retourne la liste <l> en entier.

NTHCDR peut être défini en Lisp de la manière suivante :

```
(DE NTHCDR (n l)
    (IF (<= n 0)
        l
        (NTHCDR (1- n) (CDR l))))
```

```
ex : (NTHCDR 3 '(A B C D E))  ->  (D E)
```

(NTH <n> <l>) **[SUBR à 2 arguments]**

retourne le <n>ième élément de la liste <l> le CAR de la liste étant l'élément de numéro 0. NTH est équivalent à :

(CAR (NTHCDR <n> <l>)).

NTH peut être défini en Lisp de la manière suivante :

```
(DE NTH (n l)
    (IF (<= n 0)
        (CAR l)
        (NTH (1- n) (CDR l))))
```

```
ex : (NTH 3 '(A B C D E F))    ->  D
     (NTH 10 '(A B C))         ->  NIL
```

(LAST <s>) **[SUBR à 1 argument]**

si <s> est une liste, retourne son dernier CONS. si <s> n'est pas une liste, retourne <s> lui-même.

LAST peut être défini en Lisp de la manière suivante :

```
(DE LAST (s)
    (IF (OR (ATOM s) (ATOM (CDR s)))
        s
        (LAST (CDR s))))
```

```
ex : (LAST '(A B C D E))    ->  (E)
     (LAST '(A B C . D))    ->  (C . D)
     (LAST 120)             ->  120
```

(LENGTH <s>) [SUBR à 1 argument]

si <s> est une liste, retourne le nombre d'éléments de cette liste. Si <s> n'est pas une liste (donc si <s> est un atome), LENGTH retourne 0

LENGTH peut être défini en Lisp de la manière suivante :

```
(DE LENGTH (l)
    (IF (ATOM l)
        0
        (1+ (LENGTH (CDR l)))))

ex : (LENGTH '(A (B C) D E )  ->  4
     (LENGTH 10)              ->  0
```

3.6.2 Les fonctions de création de listes

Toutes les fonctions qui vont être décrites fabriquent de nouvelles listes. La gestion de la mémoire allouée aux listes est dynamique et automatique (voir la section sur le Garbage-collector).

Du fait de la limitation extrêmement importante de l'espace adresse, Le_Lisp 80 ne peut pas construire des doublets de liste dont le CDR est un nombre entier. Si un tel cas se produit, l'erreur ERNUM se déclenche dont le libellé est :

```
** <fn> : CDR numerique : <n>
```

dans lequel <fn> est la fonction qui voulait créer le doublet et <n> est la valeur numérique.

(CONS <s1> <s2>) [SUBR à 2 arguments]

construit une liste dont le premier élément est <s1> et le reste la liste <s2>. Si <s2> est un atome, CONS produit la paire pointée

(<s1> . <s2>)

```
; En supposant que la variable FREE contient la liste des
; cellules allouables, CONS se decrit en Lisp :

    (DE CONS (x y)
        (RPLACA FREE x)
        (RPLACD (PROG1 FREE (SETQ FREE (CDR FREE))) y))

ex : (CONS 'A '(B C))        ->  (A B C)
     (CONS 1 'X)             ->  (1 . X)
```

```
(SETQ l '(X Y Z))          ->  (X Y Z)
(EQUAL (CONS (CAR l) (CDR l))
    l)                     ->  T
```

(MCONS <s1> ... <sN>) [SUBR à N arguments]

réalise un CONS multiple. L'appel de :
(MCONS s1 s2 ... sN-1 sN)
équivaut à l'appel
(CONS s1 (CONS s2 ... (CONS sN-1 sN) ...))

MCONS peut être défini en Lisp de la manière suivante :

; sous forme d'une FEXPR :

```
(DF MCONS l
    (CONS (EVAL (CAR l))
          (EVAL (IF (NULL (CDDR l))
                    (CADR l)
                    (CONS 'MCONS (CDR l))))))
```

; ou bien sous forme d'une MACRO

```
(DM MCONS (mcons . l)
    (LIST 'CONS (CADR l)
          (IF (NULL (CDDDR l))
              (CADDR l)
              (CONS 'MCONS (CDDR l)))))
```

```
ex : (MCONS 'A 'B)      ->  (A . B)
     (MCONS 'A 'B 'C)   ->  (A B . C)
```

(LIST <s1> ... <sN>) [SUBR à N arguments]

retourne la liste des valeurs des différentes expressions
<s1> ... <sN>.
En terme de CONS l'appel (LIST <s1> <s2> ... <sN-1> <sN>)
est équivalent à
(CONS <s1> (CONS <s2> ... (CONS <sN-1> (CONS <sN> NIL)) ...)).

LIST peut être défini en Lisp de la manière suivante :

```
; sous forme d'EXPR :

    (DE LIST l l)

; ou sous forme de FEXPR :

    (DF LIST l
        (LET ((r))
```

```
                (WHILE l (NEWR r (EVAL (NEXTL l))))
                r))

ex : (LIST 'A 'B 'C)    ->  (A B C)
     (LIST)             ->  NIL
```

(KWOTE <s>) [SUBR à 1 argument]

construit une liste de 2 éléments dont le 1er est l'atome QUOTE lui-même et le second la valeur de <s>.

KWOTE peut être défini en Lisp de la manière suivante :

```
(DE KWOTE (s)
    (LIST 'QUOTE s))

ex : (KWOTE 'A)                ->  (QUOTE A)
     (KWOTE (CDR '(A . B)))    ->  (QUOTE B)
```

(APPEND <l1> ... <lN>) [SUBR à N arguments]

retourne la concaténation d'une copie du premier niveau de toutes les listes <l1> ... <lN-1> à la liste <lN>. S'il n'y a qu'un argument <l1>, retourne simplement une copie du premier niveau de <l1>.

APPEND peut être défini en Lisp de la manière suivante :

```
(DE APPEND l
    (APPEND2 (CAR l)
       (IF (ATOM (CDDR l))
           (CADR l)
           (APPLY 'APPEND (CDR l)))))

; avec APPEND2 :

(DE APPEND2 (l1 l2)
    (IF (ATOM l1)
        l2
        (CONS (CAR l1) (APPEND (CDR l1) l2))))

ex : (APPEND '(A B C))           ->  (A B C)
     (APPEND '(A B C) '(X Y))    ->  (A B C X Y)
     (APPEND () '(X Y))          ->  (X Y)
```

(REVERSE <s>) [SUBR à 1 argument]

retourne une copie inversée du premier niveau de la liste <s>.

REVERSE peut être défini en Lisp de la manière suivante :

```
(DE REVERSE (s)
    (LETN REV2 ((s s) (r ()))
          (IF (ATOM s)
              r
              (REV2 (CDR s) (CONS (CAR s) r)))))
```

```
ex : (REVERSE '(A (B C) D))          ->  (D (B C) A)
```

```
; On prétend parfois que cette fonction
; inverse aussi la liste l :

(DE REV (l)
    (IF (NULL (CDR l))
        l
        (CONS (CAR (REV (CDR l)))
              (REV (CONS (CAR l)
                         (REV (CDR (REV (CDR l))))))))))
```

(COPY <l>) [SUBR à 1 argument]

fabrique une nouvelle copie de toute la liste <l>. Pour cette fonction la copie s'effectue à tous les niveaux de l'arborescence.

COPY peut être défini en Lisp de la manière suivante :

```
(DE COPY (s)
    (IF (ATOM s)
        s
        (CONS (COPY (CAR s)) (COPY (CDR s)))))
```

```
ex : (COPY 'A)                  ->  A
     (COPY '(A (B (C (D))))     ->  (A (B (C (D))))
```

```
; cette fonction ne traite pas les listes circulaires
; ou partagées. Pour cela il faut utiliser la fonction :

(DE CIRCOPY (l)
    (LET ((d))
         (LETN CIRCAUX ((l l) (new ()))
               (COND ((ATOM l) l)
                     ((CDR (ASSQ l d)))
                     (T (SETQ new (CONS NIL NIL)
                              d   (CONS (CONS l new) d))
                        (RPLAC new
                               (CIRCAUX (CAR l))
                               (CIRCAUX (CDR l)))))))))
```

(SUBST <s1> <s2> <l>) [SUBR à 3 arguments]

fabrique une nouvelle liste <l> en substituant, dans tous les niveaux de la liste, à chaque occurrence de l'expression <s2>, l'expression <s1>. La liste obtenue partage le maximum de doublets avec la liste initiale. Pour obtenir une copie complète de <l> il faut utiliser la forme :

```
(NSUBST <s1> <s2> (COPY <l>)
```

Cette fonction utilise le prédicat EQ pour réaliser le test.

SUBST peut être défini en Lisp de la manière suivante :

```
(DE SUBST (new old l)
    (COND ((EQ l old) new)
          ((ATOM l) l)
          (LET ((car (SUBST new old (CAR l)))
                (cdr (SUBST new old (CDR l))))
               (IF (AND (EQ (CAR l) car)
                        (EQ (CDR l) cdr))
                   l
                   (CONS car cdr)))))

ex : (SUBST '(X Y Z) 'A  '(A C (D A)))
     -> ((X Y Z) C (D (X Y Z)))
```

(OBLIST) [SUBR à 0 argument]

retourne la (très longue) liste de tous les symboles présents dans le système. A l'initialisation du système, cette liste contient le nom de toutes les fonctions et variables prédéfinies. Cette liste étant très longue il est conseillé d'utiliser la fonction MAPCOBLIST pour accéder successivement à tous ces symboles. Il n'est pas possible de définir simplement cette fonction en Lisp car il faut accéder aux propriétés naturelles de type A-LINK de chacun des symboles, ce qui n'est réalisable qu'avec les fonctions LOC et MEMORY.

```
ex : (OBLIST) ->

     (RMARGIN LMARGIN OUTPOS PTYPE EXPLODE TERPRI
     PRINCH OBASE READ PEEKCH READCH TEREAD ASCII
     CASCII TYPECH IMPLODE EVAL APPLY EXIT
                    .....
                    .....
                    .....
     MEMORY CALL EXECUTE END)
```

3.6.3 Les fonctions de modification physique

Toutes les fonctions qui vont être décrites dans cette section doivent être utilisées conformément au mode d'emploi, pour éviter de placer le système dans un état de confusion dramatique car elles vont permettre de modifier physiquement les structures Lisp.
En particulier il faut prendre garde à la modification physiques des listes partagées.
Toutefois la possibilité d'une véritable chirurgie sur les représentations internes des listes confère à Lisp la puissance des langages machines.

(RPLACA <l> <s>) **[SUBR à 2 arguments]**

remplace le CAR de la liste <l> par <s>. Retourne la liste ainsi modifiée <l> en valeur. Si l'argument <l> n'est pas une liste, une erreur de type se produit.

```
ex : (RPLACA '(A B C) '(X Y))  ->  ((X Y) B C)
     (RPLACA 'X 'FOO)          ->
     ** RPLACA : l'argument n'est pas une liste : X
```

(RPLACD <l> <s>) **[SUBR à 2 arguments]**

remplace le CDR de la liste <l> par <s>. Retourne la nouvelle liste <l> en valeur. Si l'argument <l> n'est pas une liste, une erreur de type se produit.

```
ex : (RPLACD '(A B C) '(X Y Z))  ->  (A X Y Z)
     (RPLACD NIL 'T)             ->
     ** RPLACD : l'argument n'est pas une liste : NIL
```

(RPLAC <l> <s1> <s2>) **[SUBR à 3 arguments]**

remplace le CAR de la liste <l> par <s1> et le CDR de la liste <l> par <s2>. Si l'argument <l> n'est pas une liste, une erreur de type apparait.

RPLAC peut être défini en Lisp de la manière suivante :

```
(DE RPLAC (l s1 s2)
    (IF (ATOM l)
        (SYSERROR 'RPLAC 'ERNLA l)
        (RPLACA l s1)
        (RPLACD l s2)))

ex : (SETQ l1 '(A B) l2 l1)     ->  (A B)
     (RPLAC l1 1 '(2))          ->  (1 2)
     l1                         ->  (1 2)
     l2                         ->  (1 2)
```

(DISPLACE <l> <ln>) [SUBR à 2 arguments]

remplace le CAR de la liste <l> par le CAR de <ln> et le CDR de <l> par le CDR de <ln>. Cette fonction est souvent utilisée dans des MACRO pour modifier physiquement l'appel de MACRO lui-même.

DISPLACE peut être défini en Lisp de la manière suivante :

```
(DE DISPLACE (l ln)
    (IF (ATOM l)
        (SYSERROR 'DISPLACE 'ERNLA l)
        (IF (ATOM ln)
            (RPLAC l 'PROGN (LIST ln))
            (RPLAC l (CAR ln) (CDR ln)))))

ex : (SETQ L1 '(A B C))        ->  (A B C)
     (SETQ L2 L1)              ->  (A B C)
     (DISPLACE L1 '(X Y))      ->  (X Y)
     L2                        ->  (X Y)
```

(PLACDL <l> <s>) [SUBR à 2 arguments]

permet d'accrocher dans le CDR de <l> un nouveau doublet dont le CAR est <s>. PLACDL retourne ce nouveau doublet. Cette étrange fonction est en fait très utilisée pour fabriquer des listes *dans le bon ordre* en un seul parcours non récursif.

PLACDL peut être défini en Lisp de la manière suivante :

```
(DE PLACDL (l s)
    (CDR (RPLACD l (CONS s ()))))

ex : (DE EVLIS (l)
         (COND ((NULL l) ())
               ((NULL (CDR l)) (LIST (EVAL (CAR l))))
               (T (LET ((tete (LIST (EVAL (CAR l)))))
                       (EVLISAUX (CDR l) tete)
                       tete)))))

     (DE EVLISAUX (rest cour)
         (WHEN rest
               (EVLISAUX (CDR rest)
                         (PLACDL cour (EVAL (CAR rest))))))
```

(NCONC <l1> ... <lN>) [SUBR à N arguments]

concatène physiquement toutes les listes <li> (c'est-à-dire place dans le CDR du dernier élément de <li-1> un pointeur sur la liste <li>). NCONC retourne la nouvelle liste <l1> en valeur. Si deux listes <li> ont la même adresse physique, NCONC construit une liste circulaire par forcage dans le dernier CDR de la liste d'un pointeur sur le 1er élément de cette même liste.

NCONC (à 2 arguments) peut être défini en Lisp de la manière suivante :

```
(DE NCONC2 (l1 l2)
    (LET ((l1 l1))
         (WHILE (CONSP (CDR l1)) (NEXTL l1))
         (RPLACD l1 l2))
    l1)

ex : (SETQ X1 '(A B C) X2 X1)  ->  (A B C)
     (NCONC X1 '(D E F))       ->  (A B C D E F)
     X2                        ->  (A B C D E F)
     (NCONC X1 X1)             ->  (A B C D E F A B C D E!
F A B ...
```

(NRECONC <l> <s>) **[SUBR à 2 arguments]**

renverse physiquement (et rapidement) la liste <l> et ajoute au moyen d'un NCONC physique la liste <s>. Cette fontion correspond donc à :
(NCONC (NREVERSE <l>) <s>)
Cette fonction doit etre manipulée avec précaution car elle modifie physiquement des structures Le_Lisp (en particulier les modifications peuvent être bouleversantes en cas de structures partagées), mais elle est évidement beaucoup plus rapide que la fonction traditionnelle REVERSE.

NRECONC peut être défini en Lisp de la manière suivante :

```
(DE NRECONC (l s)
    (IF (CONSP l)
        (NCONC (NREVERSE l) s)
      s))

ex :  (SETQ L1 '(A B C D E)) -> (A B C D E)
      (SETQ L2 (CDR L1))     -> (B C D E)
      (SETQ L3 (LAST L1))    -> (E)
      (NRECONC L1 '(X Y))    -> (E D C B A X Y)
      L1                     -> (A X Y)
      L2                     -> (B A X Y)
      L3                     -> (E D C B A X Y)
```

3.6.4 Les fonctions sur les A-listes

En Lisp, les A-listes (les listes d'association) sont des tables (au sens de SNOBOL 4) qui possèdent la structure suivante :

```
((cle1 . val1) (cle2 . val2)  ...  (cleN . valN))
```

Chaque élément d'une A-liste est un couple constitué d'une clef (le CAR de l'élément de la table) et d'une valeur (le CDR de l'élément). L'accès à une valeur est réalisé au moyen de sa clef.

Pour toutes les fonctions qui vont être décrites, l'argument <al> doit être une A-liste, et les clefs doivent être des symboles (pour les fonctions d'accès aux A-listes utilisant le prédicat EQ) ou des S-expressions quelconques (pour les fonctions utilisant le prédicat EQUAL).

(ASSQ <sym> <al>) **[SUBR à 2 arguments]**

retourne l'élément de la A-liste <al> dont la clef (le CAR) est égal au symbole <sym>, sinon ASSQ retourne NIL.

ASSQ peut être défini en Lisp de la manière suivante :

```
(DE ASSQ (at al)
    (COND ((NULL al) NIL)
          ((EQ (CAAR al) at) (CAR al))
          (T (ASSQ at (CDR al)))))
```

```
ex : (ASSQ 'B '((A) (B 1) (C D E)))  ->  (B 1)
```

(CASSQ <sym> <al>) **[SUBR à 2 arguments]**

est identique à ASSQ mais retourne la valeur seule (le CDR) de l'élément dont on précise la clef.

(CASSQ at al) est donc équivalent à (CDR (ASSQ at al))

ATTENTION : on ne peut pas distinguer la valeur NIL associée à un élément de A-liste avec l'absence de cet élément.

```
ex : (CASSQ 'C '((A) (B 1) (C D E)))  ->  (D E)
```

(SUBLIS <al> <s>) **[SUBR à 2 arguments]**

retourne une copie de l'expression <s> dans laquelle toutes les occurrences des clefs de la A-liste <al> ont été remplacées par leurs valeurs associées correspondantes. La copie retournée partage le maximum de doublets avec l'expression initiale. Cette fonction utilise le prédicat ASSQ.

SUBLIS peut être défini en Lisp de la manière suivante :

```
(DE SUBLIS (al s)
    (IF (ATOM s)
        (LET ((x (ASSQ s al))) (IF x (CDR x) s))
        (LET ((car (SUBLIS al (CAR s)))
              (cdr (SUBLIS al (CDR s))))
             (IF (AND (EQ car (CAR s))
                      (EQ cdr (CDR s)))
                 s
                 (CONS car cdr)))))

ex :  (SUBLIS '((A . Z) (B 2 3)) '(A (B A C) D B . B)))
        -> (Z ((2 3) Z C) D (2 3) 2 3)
```

3.7 Les fonctions sur les symboles

3.7.1 Les fonctions d'accès aux symboles

L'accès aux valeurs des symboles est normalement réalisé par EVAL à l'évaluation d'un symbole. EVAL de plus teste si une valeur a été affectée à ce symbole : dans le cas d'un symbole indéfini il provoque une erreur dont le libellé est :

```
** EVAL : variable indefine : <sym>
```

(BOUNDP <sym>) **[SUBR à 1 argument]**

l'argument <sym> doit être un symbole. BOUNDP teste si ce symbole possède une valeur et retourne T si ce test est vérifié et NIL dans le cas contraire. Cette fonction permet d'éviter l'erreur : *variable indéfinie*. Du fait de la représentation des valeurs indéfinies il n'est pas possible de décrire cette fonction en Lisp.

```
ex : (BOUNDP T)          ->  T
     (BOUNDP NIL)        ->  T
     (BOUNDP 'foofoo)    ->  NIL
                 ; si foofoo n'a pas de valeur
```

(CVAL <sym> <s>) **[SUBR à 1 ou 2 arguments]**

l'argument <sym> doit être un symbole. Si le 2ème argument est présent il devient la nouvelle valeur de ce symbole. CVAL retourne dans tous les cas la valeur de ce symbole. Cette fonction est équivalente à la fonction EVAL qui évaluerait un symbole mais ne teste pas la validité de la valeur de ce symbole. Cette fonction permet en autre de décrire la fonction EVAL elle-même.

3.7.2 Les fonctions de modification des symboles

(MAKUNBOUND <sym>) **[SUBR à 1 argument]**

change la valeur du symbole <sym> de telle manière que tout nouvel accès à sa valeur déclenche l'erreur : variable indéfinie. MAKUNBOUND retourne <s> en valeur.

```
ex : (SETQ x 10)        ->  10
     x                  ->  10
     (MAKUNBOUND 'x)    ->  x
     x
     ** EVAL : variable indefinie : X
```

(SET <sym1> <s1> ... <symN> <sN>) **[SUBR à N arguments]**

change la valeur des symboles <symi> par les valeurs respectives de <si>. SET retourne en valeur <sN>.

```
ex : (SET 'X '(X Y))      ->   (X Y)
     X                    ->   (X Y)
```

(SETQ <sym1> <s1> ... <symN> <sN>) **[FSUBR]**

<sym1> ... <symN> sont des symboles qui ne sont pas évalués ;
<s1> ... <sN> sont des expressions quelconques qui seront évaluées par la fonction SETQ elle-même. SETQ est la fonction d'affectation la plus utilisée : chaque symbole <symi> est initialisé avec la valeur de l'expression correspondante <si>. Ces affectations ainsi que les évaluations des expressions <ei> sont réalisées *en séquence*. SETQ retourne <sN> en valeur.

```
ex : (SETQ L1 '(A B C))     ->  (A B C)
     (SETQ L2 L1)           ->  (A B C)
     (SETQ L3 L2 L4 'FOO)   ->  FOO
     L3                     ->  (A B C)
```

(NEXTL <sym>) **[FSUBR]**

<sym> (qui n'est pas évalué) doit être un symbole dont la valeur doit être une liste. NEXTL retourne le CAR de cette liste en valeur et donne comme nouvelle valeur de <sym> le CDR de son ancienne valeur. Cette fonction est très utile pour *avancer dans une liste* qui est la valeur d'un symbole.

NEXTL peut être défini en Lisp de la manière suivante :

```
(PROG1 (CAR <sym>) (SETQ <sym> (CDR <sym>)))
```

```
ex : (SETQ A '(X Y Z))    ->  (X Y Z)
     (NEXTL A)            ->  X
     A                    ->  (Y Z)
```

(NEWL <sym> <s>) **[FSUBR]**

<sym> (qui n'est pas évalué) doit être un symbole dont la valeur est une liste. NEWL place en tête de cette liste la valeur de <s> et retourne en valeur la nouvelle liste formée. L'utilisation conjuguée des fonctions NEXTL et NEWL permet de construire très naturellement des piles-listes.

; cette fonction correspond en Lisp à la forme :

```
(SETQ <sym> (CONS <s> <sym>))
```

```
ex : (SETQ A '(X Y Z))    ->  (X Y Z)
     (NEWL A 'W)          ->  (W X Y Z)
     A                    ->  (W X Y Z)
```

(NEWR <sym> <s>) **[FSUBR]**

<sym> (qui n'est pas évalué) doit être un symbole dont la valeur est une liste. NEWR ajoute en queue de cette liste la valeur de <s> et retourne en valeur la nouvelle liste formée. Si la valeur de <sym> n'est pas une liste, NEWR crée pour nouvelle valeur de <sym> la liste composée de l'élément <s>.

NEWR peut être défini en Lisp de la manière suivante :

```
(DF NEWR (sym s)
    (IF (CONSP (CVAL sym))
        (NCONC (CVAL sym) [(EVAL s)])
        (SET sym [(EVAL s)])))
```

```
ex : (SETQ A '(X Y Z))    ->  (X Y Z)
     (NEWR A 'W)          ->  (X Y Z W)
     A                    ->  (X Y Z W)
```

```
(SETQ B ())          ->  ()
(NEWR B 'Z)          ->  (Z)
B                    ->  (Z)
```

(INCR <sym> <n>) **[FSUBR]**

<sym> doit être le nom d'un symbole qui possède une valeur numérique. INCR va incrémenter cette valeur de la valeur de l'expression <n> si celle-ci est donnée ou de 1 dans le cas contraire.

(INCR <sym> <n>) est équivalent à : (SETQ <sym> (+ <sym> <n>))
et (INCR <sym>) est équivalent à : (SETQ <sym> (1+ <sym>))

INCR peut être défini en Lisp de la manière suivante :

```
(DF INCR (var val)
    (SET var
         (IF val
             (+  (EVAL var) (EVAL val))
             (1+ (EVAL var)))))
```

(DECR <sym> <n>) **[FSUBR]**

<sym> doit être le nom d'un symbole qui possède une valeur numérique. DECR va décrémenter cette valeur de la valeur de l'expression <n> si celle-ci est donnée ou de 1 dans le cas contraire.

(DECR <sym> <n>) est équivalent à : (SETQ <sym> (- <sym> <n>))
et (DECR <sym>) est équivalent à : (SETQ <sym> (1- <sym>))

DECR peut être défini en Lisp de la manière suivante :

```
(DF DECR (var val)
    (SET var
         (IF val
             (-  (EVAL var) (EVAL val))
             (1- (EVAL var)))))
```

3.7.3 Les fonctions sur les P–Listes

En Lisp, les P-LIST (listes de propriétés) sont des listes constituées d'indicateurs et de valeurs qui ont la structure suivante :

(<indic1> <val1> <indic2> <val2> ... <indicN> <valN>)

A chaque indicateur <indici> est associée une valeur <vali> qui est l'élément suivant de la P-LIST. Les recherches sur les P-LIST s'effectuent donc deux éléments par deux éléments.

Chaque symbole possède une P-LIST qui lui est propre.

Les arguments des fonctions sur les P-LIST sont :
<pl> - un symbole dont on veut utiliser la P-LIST .
Si <pl> est un nombre, une liste ou bien l'atome NIL, une erreur de type est déclenchée.
<ind> un indicateur. Celui-ci doit être un symbole ou un nombre, la recherche des indicateurs utilisant le prédicat EQ.
<pval> peut être n'importe quelle expression.

(PLIST <pl> <s>) **[SUBR à 1 ou 2 arguments]**

si le deuxième argument <s> est fourni il devient la nouvelle P-LIST du symbole <pl>. PLIST retourne dans tous les cas la P-LIST associée au symbole <pl>. Si le symbole <pl> ne possède pas de P-LIST, PLIST retourne NIL.

PLIST ne peut pas être défini simplement en Lisp.

```
ex :   (PLIST 'rose '(nom commun genre feminin))
           ->  (nom commun genre feminin)
       (PLIST 'rose)  ->  (nom commun genre feminin)
```

(GETPROP <pl> <ind>) **[SUBR à 2 arguments]**

retourne la valeur associée à l'indicateur <ind> dans la P-LIST du symbole <pl>. Si l'indicateur n'existe pas, GETPROP retourne NIL.

ATTENTION : on ne peut pas discerner la valeur égale à NIL d'un indicateur avec l'absence de cet indicateur.

GETPROP peut être défini en Lisp de la manière suivante :

```
(DE GETPROP (pl ind)
    (LET ((pl (PLIST pl)))
         (COND ((NULL pl) NIL)
               ((EQ (CAR pl) ind) (CADR pl))
               (T (SELF (CDDR pl))))))
```

```
ex : (PLIST 'rose)              ->  (nom commum genre feminin)
     (GETPROP 'rose 'genre)     ->  feminin
     (GETPROP 'rose 'famille)   ->  NIL
```

(ADDPROP <pl> <pval> <ind>) [SUBR à 3 arguments]

rajoute en tête de la P-LIST <pl> l'indicateur <ind> et sa valeur associée <pval>. ADDPROP retourne <pl> en valeur.

ADDPROP peut être défini en Lisp de la manière suivante :

```
(DE ADDPROP (pl pval ind)
    (PLIST pl (CONS ind (CONS pval (PLIST pl))))
    pl)
```

```
ex : (PLIST 'PLT)               ->  (I1 A I2 B)
     (ADDPROP 'PLT 'C 'I1)      ->  'PLT
     (PLIST 'PLT)               ->  (I1 C I1 A I2 B)
```

(PUTPROP <pl> <pval> <ind>) [SUBR à 3 arguments]

si l'indicateur <ind> existe déjà sur la P-LIST <pl>, sa valeur associée prend la nouvelle valeur <pval>, sinon l'indicateur <ind> et sa valeur associée <pval> sont ajoutés en tête de <pl> (d'une manière identique à la fonction ADDPROP). PUTPROP retourne <pl> en valeur.

PUTPROP peut être défini en Lisp de la manière suivante :

```
(DE PUTPROP (pl pval ind)
    (LET ((p (PLIST pl)))
         (COND ((NULL p) (ADDPROP pl pval ind))
               ((EQ (CAR p) ind)
                  (RPLACA (CDR p) pval))
               (T (SELF (CDDR p)))))
    pl)
```

```
ex : (PLIST 'PLT)               ->  (I1 A I2 B)
     (PUTPROP 'PLT 'C 'I1)      ->  PLT
     (PLIST 'PLT)               ->  (I1 C I2 B)
     (PUTPROP 'PLT 0 'I9)       ->  PLT
     (PLIST 'PLT)               ->  (I9 0 I1 C I2 B)
```

(REMPROP <pl> <ind>) [SUBR à 2 arguments]

enlève de la P-LIST <pl> l'indicateur <ind> s'il existe ainsi que sa valeur associée. REMPROP retourne <pl> en valeur.
L'utilisation conjuguée des fonctions REMPROP et ADDPROP permet d'utiliser les P-LIST comme des piles de propriétés-valeurs.

REMPROP peut être défini en Lisp de la manière suivante :

```
(DE REMPROP (pl ind)
    (LET ((pll pl) (pl2 (PLIST pl)))
         (COND ((NULL pl2) ())
               ((EQ (CAR pl2) ind)
                  (RPLACD pll (CDDR pl2)))
               (T (SELF pl2 (CDDR pl2)))))
    pl)

ex : (PLIST 'PLT)          ->  (I1 A I2 B)
     (REMPROP 'PLT 'I2)    ->  PLT
     (PLIST 'PLT)          ->  (I1 A)
```

3.8 Les fonctions sur les P–Names

Rappelons que :
- le P-NAME d'un symbole est son nom externe
- le P-NAME d'un nombre est sa représentation externe dans la base de conversion de sortie courante (c'est-à-dire OBASE).

Toutes ces fonctions peuvent être décrites au moyen des 2 fonctions de base, IMPLODE et EXPLODE qui permettent de passer de la représentation interne d'un P-NAME à sa représentation externe sous forme de liste de caractères et vice-versa. Ces 2 fonctions vont elles-mêmes pouvoir se décrire en Lisp au moyen d'une redirection des flux d'entrée sortie. Voir les fonctions BOL et EOL du chapitre suivant.

(EXPLODE <s>) **[SUBR à 1 argument]**

retourne la liste de tous les codes ASCII de la représentation externe de l'expression <s>, qui peut être de type quelconque. EXPLODE retourne la liste des codes qui seraient imprimés si on demandait l'impression de <s> au moyen de la fonction PRIN (du reste il n'y a pas de fonction EXPLODE à proprement parler dans l'interprète mais un changement de direction du flux de sortie produit par la fonction PRIN).

```
ex : (EXPLODE -120)            ->  (45 49 50 48)
     (EXPLODE '(CAR '(A B)))
        ->  (40 67 65 82 32 39 40 65 32 66 41 41)
```

(IMPLODE <ln>) **[SUBR à 0 ou 1 argument]**

Si l'argument <ln> est fourni, IMPLODE suppose que c'est une liste de codes ASCII. IMPLODE retourne l'objet Lisp qui possède comme représentation externe la liste des codes internes <ln>. IMPLODE est l'inverse de EXPLODE et permet de fabriquer de nouveaux objets Lisp au moyen de leur représentation externe d'une manière analogue à la fonction READ (il n'y a pas de fonction spéciale IMPLODE dans l'interprète mais simplement un changement d'origine du flux d'entrée de la fonction READ). Si l'argument n'est pas fourni, IMPLODE

va utiliser la propre valeur du symbole IMPLODE. Enfin, si la liste est trop longue, le reste de la liste est remis dans le symbole IMPLODE.

```
ex : (IMPLODE '(45 50 51 55))    -> -237
     (IMPLODE (EXPLODE '(A B)))  -> (A B)
```

(PLENGTH <sym>) [SUBR à 1 argument]

retourne en valeur le nombre de caractères du P-NAME du symbole <sym>. Si l'argument <sym> n'est pas un symbole, PLENGTH retourne 0.

PLENGTH peut être défini en Lisp de la manière suivante :

```
(DE PLENGTH (at)
    (IF (SYMBOLP at)
        (LENGTH (EXPLODE at))
        0))

ex : (PLENGTH 'FOOBAR)  -> 6
     (PLENGTH ())       -> 3  ; taille de l'atome NIL
     (PLENGTH 120)      -> 0
```

(CONCAT <at1> ... <atN>) [SUBR à N arguments]

fabrique un nouveau symbole dont le P-NAME est construit en concaténant tous les P-NAMEs des différents atomes <at1> ... <atN>.

CONCAT peut être défini en Lisp de la manière suivante :

```
(DE CONCAT l
    (IMPLODE [#/|  (MAPCAN 'EXPLODE l) #/|]))

ex : (CONCAT 'foo (1+ 25) 'bar)   ->  FOO26BAR

; voici comment décrire un générateur de symboles

(SETQ GENSYM-COUNTER 100)

(DE GENSYM ()
    (CONCAT 'G (INCR GENSYM-COUNTER)))
```

(CHRPOS <cn> <sym>) [SUBR à 2 arguments]

retourne en valeur la position du caractère de code ASCII <cn> dans le P-NAME du symbole <sym>. Si le code <cn> n'existe pas dans le P-NAME de l'atome CHRPOS retourne NIL.

CHRPOS peut être défini en Lisp de la manière suivante :

```
(DE CHRPOS (c at)
    (LET ((l (EXPLODE at)) (n 0))
         (COND ((NULL l) NIL)
```

```
                    ((EQ (CAR l) c) n)
                    (T (SELF (CDR l) (1+ n))))))

ex : (CHRPOS #/Y '|OTYoty|)            -> 2
     (CHRPOS #/N '|OTY|)               -> NIL
     (CHRPOS #/D '|0123456789ABCDEF|)  -> 13
```

(CHRNTH <n> <sym>) **[SUBR à 2 arguments]**

retourne le code interne du <n>ième caractère du P-NAME du symbole <sym> et s'il n'existe pas retourne la valeur NIL. La position (le rang) du premier caractère est 0 (et non 1).

CHRNTH peut être défini en Lisp de la manière suivante :

```
(DE CHRNTH (n at)
    (WHEN (SYMBOLP at)
          (NTH n (EXPLODE at))))

ex : (CHRNTH 0 'FOO)                    -> 70   i.e. #/F
     (CHRNTH 10 '|0123456789ABCDEF|)    -> 61   i.e. #/A
```

(ALPHALESSP <sym1> <sym2>) **[SUBR à 2 arguments]**

retourne T si le P-NAME du symbole <sym1> est inférieur ou égal (lexicographiquement) au P-NAME du symbole <sym2>, sinon retourne NIL. Cette fonction est utilisée pour réaliser des tris alphabétiques.

ALPHALESSP peut être défini en Lisp de la manière suivante :

```
(DE ALPHALESSP (a1 a2)
    (LET ((l1 (EXPLODE a1)) (l2 (EXPLODE a2)))
         (COND ((NULL l1) T)
               ((NULL l2) NIL)
               ((= (CAR l1) (CAR l2))
                (SELF (CDR l1) (CDR l2)))
               ((< (CAR l1) (CAR l2))))))

ex : (ALPHALESSP 'A 'A)       -> T
     (ALPHALESSP 'B 'A)       -> NIL
     (ALPHALESSP 'A 'B)       -> T
     (ALPHALESSP 'ZZZ 'ZZZZ)  -> T
```

Comment réaliser le tri d'une liste de symboles par échange physique :

```
(DE SORTL  (l)
    (LET ((s (APPEND l)))
         (MAP (LAMBDA (sl1)
                (IF (CDR l)
                    (MAP (LAMBDA (sl2)
                           (IF (ALPHALESSP (CAR sl1) (CAR sl2))
                               ()
```

```
                                    (RPLACA s12
                                       (PROG1 (CAR s11)
                                          (RPLACA s11 (CAR s12))))))
                              (CDR s11))))
                    s)
               s))
```

Voici une autre manière de trier une liste de symboles au moyen d'un tri-fusion récursif :

```
(DE SORTL (l)
    ; tri la liste l
    (IF (NULL (CDR l))
        l
        (FUSION (SORTL (MOITIE1 l))
                (SORTL (MOITIE2 l)))))

(DE FUSION (l1 l2)
    ; fusionne les 2 listes triees l1 et l2
    (COND
        ((NULL l1) l2)
        ((NULL l2) l1)
        ((ALPHALESSP (CAR l1) (CAR l2))
            (CONS (CAR l1) (FUSION (CDR l1) l2)))
        (T  (CONS (CAR l2) (FUSION l1 (CDR l2))))))

(DE MOITIE1 (l)
    ; retourne la 1ere moitie de l
    (FIRSTN (// (LENGTH l) 2) l))

(DE MOITIE2 (l)
    ; retourne la 2eme moitie de l
    (NTHCDR (// (LENGTH l) 2) l))
```

Cet exemple n'est pas optimum quant au nombre de CONS effectués. L'utilisation des fonctions de modification physique des listes permet la suppression de tous les CONS des fonctions FUSION et MOITIE1. Voici une version rapide du tri récursif :

```
(DE SORTL (L)
    (IF (NULL (CDR L))
        L
        (LET ((L1 L)
              (LET ((L (NTHCDR (1+ (// (LENGTH L) 2)) L)))
                   (LET ((L2 (PROG1 (CDR L) (RPLACD L ()))))
                        (FFUSION (SORTL L1) (SORTL L2)))))))


(DE FFUSION (l1 l2)
    ; fusionne physiquement les 2 listes triees
    (WHEN (ALPHALESSP (CAR L2) (CAR L1))
```

```
        (SETQ l1 (PROG1 l2 (SETQ l2 l1))))
(LET ((r l1) (l1 (CDR l1)) (l2 l2))
     (COND ((NULL l1) (RPLACD r l2))
           ((NULL l2) (RPLACD r l1))
            ((ALPHALESSP (CAR l1) (CAR l2))
               (RPLACD r l1)
               (SELF l1 (CDR l1) l2))
            (T (RPLACD r l2)
               (SELF l2 l1 (CDR l2)))))
l1)
```

3.9 Les fonctions sur les caractères

Ces fonctions vont manipuler les codes internes des caractères <cn>.

(ASCII <cn>) **[SUBR à 1 argument]**

retourne le caractère de code ASCII <cn> (modulo 256).

```
ex : (ASCII 67)          ->  C
     (1+ (ASCII 49))     ->  2
```

(CASCII <ch>) **[SUBR à 1 argument]**

retourne le code ASCII du caractère <ch>. Un caractère étant un symbole ou un nombre dont le P-LEN est égal à 1. Il existe des #-macros qui permettent de rentrer des constantes ASCII (voir la section sur les #-macros).

```
ex : (CASCII 'C)  ->  67
     (CASCII 1)   ->  49
```

3.10 Les fonctions sur les nombres

Toutes les fonctions de cette section testent le type de leurs arguments qui doivent être des nombres. Si ce type est erronné l'erreur ERNNA se déclenche dont le libellé est :

```
** <fn> : l'argument n'est pas un nombre : <n>
```

De même en cas de débordement arithmétique l'erreur ERARI se déclenche dont le libellé est :

```
** <fn> : debordement numerique : <n>
```

3.10.1 Les fonctions arithmétiques

(1+ <n>) **[SUBR à 1 argument]**

retourne la valeur : <n> + 1 .

```
ex :  (1+ 6)     ->  7
      (1+ -3)    ->  -2
```

(1- <n>) **[SUBR à 1 argument]**

retourne la valeur : <n> - 1

```
ex :  (1- 7)     ->  6
      (1- -3)    ->  -4
```

(+ <n1> ... <nN>) **[SUBR à N arguments]**

retourne la valeur de la somme : <n1> + <n2> + ... + <nN>. Si aucun argument n'est fourni, + retourne l'élément neutre de l'addition 0.

```
ex :  (+ 5 6 7)     ->  18
      (+ 5 6)       ->  11
      (+ 8)         ->  8
      (+)           ->  0
```

(- <n1> ... <nN>) [SUBR à N arguments]

retourne la valeur de la différence : <n1> - <n2> - ... - <nN>. Si aucun argument n'est fourni, - retourne l'élément neutre à droite de la soustraction, 0. Si cette fonction ne possède qu'un argument elle réalise une négation arithmétique.

```
ex : (- 8 3 2)      -> 3
     (- 6 2)        -> 4
     (- 12)         -> -12
     (-)            -> 0
```

(* <n1> ... <nN>) [SUBR à N arguments]

retourne la valeur du produit : <n1> * <n2> * ... * <nN>. Si aucun argument n'est fourni, * retourne l'élément neutre de la multiplication, 1.

```
ex : (* 5 6 7)      -> 210
     (* 10 4)       -> 40
     (* 100)        -> 100
     (*)            -> 1
```

(/ <n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du quotient de : <n1> / <n2>. Si l'argument <n2> est égal à 0, une erreur se produit. Dans certaines versions de Le_Lisp, la caractère / est un caractère spécial qui permet de quoter des caractères en entrée (voir les fonctions d'entrée). Dans ce cas il faut soit quoter ce caractère lui-même au moyen d'un deuxième slash //, soit utiliser la notation |</|.

```
ex : (/ 20 5)       -> 4
     (/ 10 (1- 1))  -> ** erreur division par 0
```

; et si / est un caractère spécial

```
     (// 100 5)     -> 20     ou bien
     (|/| 100 5)    -> 20
```

(\<n1> <n2>) [SUBR à 2 arguments]

retourne la valeur du reste de la division de <n1> par <n2>.

```
ex : (\ 11 3)       -> 2
     (\ 12 4)       -> 0
```

(ABS <n>) **[SUBR à 1 argument]**

retourne la valeur absolue de l'argument <n>, |<n>|.

```
ex : (ABS 10)   ->  10
     (ABS -10)  ->  10

     (* x (ABS x))  ; calcule le carre de x en conservant
                    ; son signe, dit G. Baudet.
```

(SCALE <n1> <n2> <n3>) **[SUBR à 3 arguments]**

retourne la valeur du calcul <n1> * <n2> / <n3>
(SCALE <n1> <n2> <n3>) est donc équivalent à :
(/ (* <n1> <n2>) <n3>)
La différence réside dans le fait que le résultat intermédiaire est calculé en double précision.

```
ex :  (SCALE 100 100 100)   ->  100
```

3.10.2 Les tests sur les nombres

(= <n1> <n2>) **[SUBR à 2 arguments]**

si <n1> = <n2> alors = retourne <n1>, sinon = retourne NIL.

```
ex : (= 10 10)      ->  10
     (= -3 3)       ->  NIL
```

(<> <n1> <n2>) **[SUBR à 2 arguments]**

si <n1> est différent de <n2> alors <> retourne <n1>, sinon <> retourne NIL.

```
ex : (<> 11 10)     ->  11
     (<> -3 -3)     ->  NIL
```

(>= <n1> <n2>) **[SUBR à 2 arguments]**

si <n1> >= <n2> alors >= retourne <n1> sinon >= retourne NIL.

```
ex : (>= 3 7)       ->  NIL
     (>= 7 7)       ->  7
```

(> <n1> <n2>) **[SUBR à 2 arguments]**

si <n1> > <n2> alors > retourne <n1> sinon > retourne NIL.

```
ex : (> 5 5)      ->  NIL
     (> 7 4)      ->  7
```

(<= <n1> <n2>) **[SUBR à 2 arguments]**

si <n1> <= <n2> alors <= retourne <n1> sinon <= retourne NIL.

```
ex : (<= 5 5)     ->  5
     (<= 4 6)     ->  4
```

(< <n1> <n2>) **[SUBR à 2 arguments]**

si <n1> < <n2> alors < retourne <n1> sinon < retourne NIL.

```
ex : (< 5 5)      ->  NIL
     (< 4 5)      ->  4
```

3.10.3 Les fonctions booléennes

Pour toutes les fonctions qui vont être décrites, les arguments **<n1>** et **<n2>** doivent être des octets. De plus ces fonctions ne provoquent jamais de débordement numérique.

(LOGAND <n1> <n2>) **[SUBR à 2 arguments]**

effectue l'opération de ET logique entre <n1> et <n2>.

```
ex : (LOGAND #$36 #$25)   ->  #$24
```

; pour savoir si <n> est une puissance de 2 évaluez :

```
(= <n> (LOGAND <n> (- <n>)))
```

(LOGOR <n1> <n2>) **[SUBR à 2 arguments]**

effectue l'opération de OU logique entre <n1> et <n2>.

```
ex : (LOGOR #$15 #$17)   ->  #$17
```

(LOGXOR <n1> <n2>) **[SUBR à 2 arguments]**

effectue l'opération de OU exclusif logique entre <n1> et <n2>.

```
ex : (LOGXOR 5 3)   ->  6
```

CHAPITRE 4

LES ENTREES/SORTIES

Les entrées/sorties s'effectuent au niveau du caractère, de la ligne ou de la S-expression Lisp, sur des flux séquentiels.

Toutefois de nombreuses fonctions ont été rajoutées pour pouvoir utiliser tous les périphériques spéciaux connectés au système.

Il existe deux façons de représenter des caractères :

- sous forme d'atome mono-caractère (nombre ou symbole) <ch>
- sous forme de nombre représentant le code interne (ASCII) de ces caractères <cn>

Le_Lisp utilise l'une ou l'autre de ces représentations. Les fonctions qui utilisent la première représentation sont en général suffixée par CH et celles utilisant la représentation sous forme de code ASCII par CN. Cette deuxième représentation est toujours utilisée en cas de liste de caractères <lcn>.

4.1 Les fonctions d'entrée de base

Toutes les lectures sont réalisées dans le flux d'entrée qui est associé soit au terminal, soit à un fichier sélectionné par la fonction INPUT.

Voici les fonctions d'entrée de base.

(READ) [SUBR à 0 argument]

lit la S-expression suivante de type quelconque (atome ou liste) du flux d'entrée et la retourne en valeur. READ est la principale fonction de lecture et permet de lire des objets Lisp. Son fonctionnement détaillé est donné dans la section suivante.

Si une erreur de syntaxe Lisp est détectée, l'erreur ERSXT se déclenche dont le libellé est :

```
** <fn> : erreur de syntaxe : <n>
```

dans lequel le nom de la fonction qui a provoqué l'erreur est imprimé (READ ou IMPLODE) ainsi que le type codé de cette erreur. Voici la liste des codes d'erreurs :

1 : P-NAME de symbole trop long (> 62 caractères).
3 : une S-expression ne débute pas par (ou par un symbole.
4 : mauvaise construction . <s>)
5 : liste de IMPLODE incorrecte.
6 : liste de IMPLODE trop petite.
10 : #-macro inconnue
11 : dans #$ mauvais chiffre

Le plus souvent cette erreur provient d'une mauvaise utilisation des paires pointées ou d'un P-NAME trop long (de plus de 62 caractères), d ˜ à l'oubli du caractère | .

(READCH) [SUBR à 0 argument]

lit et retourne en valeur le caractère suivant du flux d'entrée. Ce caractère est retourné sous la forme d'un atome (symbole pour une lettre et nombre pour un chiffre).
Cette fonction n'effectue pas de conversion automatique des caractères minuscules en caractères majuscules.

(READCN) [SUBR à 0 argument]

lit et retourne en valeur le caractère suivant du flux d'entrée. Ce caractère est retourné sous la forme d'un code interne (c'est-à-dire d'un nombre). Comme pour la fonction précédente, il n'y a pas de conversion automatique des caractères minuscules en caractères majuscules. Cette fonction est à préférer à la fonction précédente car elle est plus rapide et ne crée pas de nouveaux symboles.

(PEEKCH) [SUBR à 0 argument]

retourne en valeur le caractère suivant du flux d'entrée d'une manière identique à la fonction READCH, toutefois ce caractère n'est pas véritablement lu mais seulement *consulté*. Il en résulte que des appels successifs de la fonction PEEKCH retournent toujours le même résultat. PEEKCH est utilisé pour tester le caractère suivant du flux d'entrée avant de le lire véritablement.

(PEEKCN) [SUBR à 0 argument]

est identique à la fonction précédente, mais retourne un code ASCII. Cette fonction est à préférer à la fonction précédente car elle est plus rapide et ne crée pas de nouveaux symboles.

4.2 Le contrôle des fonctions d'entrée

4.2.1 Le tampon d'entrée

Toutes les fonctions d'entrée que nous avons vu, accèdent aux différents caractères à travers un *tampon d'entrée*. Ce tampon est accessible à l'utilisateur au moyen de fonctions spécialisées bien évidemment redéfinissables.

(INBUF <n> <cn>) **[SUBR à 1 ou 2 arguments]**

considère le tampon d'entrée comme un vecteur et permet d'avoir accès et/ou de modifier le <n>ième caractère du tampon d'entrée avec le code interne <cn> si l'argument <cn> est donné. INBUF retourne toujours en valeur la <n>ième caractère du tampon d'entrée.

(INMAX <n>) **[SUBR à 0 ou 1 argument]**

permet de spécifier, si <n> est fourni, le nombre maximum de caractères disponibles dans le tampon d'entrée. Si <n> n'est pas fourni, INMAX retourne le nombre actuel de caractères du tampon d'entrée.

(BOL) **[SUBR à 0 argument]**

quand il n'y a plus de caractères à lire dans le tampon d'entrée (c'est-à-dire lorsqu'on a lu les INMAX caractères du tampon), le système déclenche une interruption programmée de nom BOL. Normalement cette fonction remet dans le tampon d'entrée la ligne suivante lue sur le terminal ou sur un fichier. Mais il est possible de redéfinir temporairement cette fonction pour remplir explicitement le tampon d'entrée et positionner INMAX.

Voici par exemple comment il est possible de définir la fonction IMPLODE en Lisp par redéfinition dynamique de BOL :

```
(DE IMPLODE (l)
    ; 'l' est une liste de codes ascii
    (FLET ((BOL ()
                (IF (NULL l)
                    (SYSERROR 'IMPLODE 'ERSXT 6)
                    (IFN (NUMBERP (CAR l))
                         (SYSERROR 'IMPLODE 'ERSXT 5)
                         (INBUF 0 (NEXTL l))
                         (INMAX l)))))
          (READ)))
```

4.2.2 L'utilisation du terminal en entrée

A chaque demande de ligne sur le terminal, le système imprime un *caractère d'invite* qui peut être modifié au moyen de la fonction PROMPT. De plus un éditeur de ligne minimum est activé qui interprète les caractères suivants :

RUBOUT ou DELETE ou BACKSPACE ou flèche à gauche : détruit le dernier caractère entré.

^U ou ^X : détruit toute la ligne entrée

Il existe dans le système Le_Lisp un véritable éditeur très puissant et extensible, EMACS, dont une version réduite se nomme PEPE dans le système Le_Lisp 80, inclu d'une façon standard, qui permet d'éditer des fichiers sur le disque. Un chapitre lui est consacré.

(PROMPT <sym>) **[SUBR à 0 ou 1 argument]**

permet de connaître ou de modifier le caractère d'invite imprimé par le système quand celui-ci est en attente de caractères lus sur le terminal. Par défaut ce caractère est le ?

```
ex : ? ()          ; l'invite normale de lecture
     = NIL
     ? (prompt '|> |)
     = >
     > ()          ; l'invite est changee
     = ()
     ? (prompt '|allo>|)
     allo>         ; l'invite est un vrai symbole.
```

4.2.3 La lecture standard

La lecture des S-expressions Lisp par la fonction READ s'effectue en *format libre :* chaque élément syntaxique peut être encadré d'un ou plusieurs espaces.

Durant la lecture des symboles seuls les 62 premiers caractères sont pris en compte. Les caractères de contrôle (c'est-à-dire les caractères entrés en pressant simultanément la touche CTRL et le caractère) sont imprimés *à la DEC* (c'est-à-dire ^ suivi du caractère). S'il faut insérer des caractères spéciaux dans un symbole, il faut soit les faire précéder du caractère spécial quote-caractère / (le slash par défaut) soit encadrer tous les caractères du symbole avec le caractère délimiteur de symboles (la barre de valeur absolue | par défaut).

(ICASE <i>) **[SUBR à 0 ou 1 argument]**

cette fonction permet de contrôler la conversion automatique des caractères minuscules en leur équivalent majuscule. Si la valeur de l'argument <i> est NIL la conversion est automatique, si elle est différente de NIL aucune conversion n'est réalisée.

```
ex : CaR                 correspond a l'atome   CAR
     LONGTRESLONGATOME     "        "           LONGTRESLONGATOME
     RE/(3/)               "        "           RE(3)
     |Foo Bar|             "        "           Foo Bar

     ; mais si on lit les symboles de la maniere suivante :

     (BIND ((ICASE T))
           (READ))

     CdR                 correspond a l'atome     CdR
     |Ach|                  "        "           Ach
```

Les nombres entiers sont représentés par une suite de chiffres décimaux qui peut être précédée du signe + ou du signe -.

Il existe d'autres manières de rentrer des nombres en particulier grâce aux #-MACRO qui permettent de décrire des nombres dans n'importe quelle base, ou de rentrer des codes internes de caractères ASCII.

```
ex :  123          correspond au nombre     123
      +27               "        "          27
      -45               "        "          -45
      +            n'est pas un nombre
```

Il est possible d'insérer des commentaires, qui sont totalement ignorés au cours de la lecture. Un commentaire est une suite de caractères quelconques précédée d'un caractère de type début de commentaire et terminée par le caractère de type fin de commentaires. Par défaut il n'y a qu'un caractère de type début de commentaires, le caractère

point-virgule (;) et qu'un seul caracrère fin de commentaires le caractère Return. Par défaut tout commentaire s'arrêtera donc en fin de ligne.

```
ex : (DE FOO (N              ; le nombre
             L)              ; la liste
       (IF (= N 0)           ; plus rien a faire
           ...
```

La représentation des listes est classique : une liste se représente par une parenthèse ouvrante (suivie des éléments de la liste séparés par des espaces et suivis d'une parenthèse fermante). Il est possible également d'utiliser la notation pointée généralisée :

```
( S-expr   .   S-expr )

ex :  (A . (B . (C . D))) correspond a (A B C . D)
      ((A) . (B))              "       "   ((A) B)
```

A la fin d'une expression lue par la fonction READ, il peut y avoir un nombre quelconque de parenthèses fermantes qui sont ignorées. Cela permet de refermer à coup sûr les S-expressions avec une giclée de parenthèses fermantes sans avoir à les dénombrer.

```
(DE FOO (n) (ADD1 n))))))    ; est lu sans erreur
```

4.2.4 Les types des caractères

L'analyseur lexical Le_Lisp (c'est-à-dire la fonction READ) utilise une *table de lecture* pour effectuer commodément son analyse. Cette table associe un type codé à chacun des caractères.

Cette table de lecture est totalement accessible à l'utilisateur qui peut ainsi la changer pour pouvoir lire facilement de nouveaux dialectes de Lisp aux syntaxes étranges (venues d'ailleurs?).

Les types de caractères disponibles sont les suivants :

0 : type NULL. Tous les caractères de ce type sont complètement ignorés à la lecture (Ex: le caractère Line/Feed, le caractère *Avance–bande* ou NULL ...).

1 : type QUOTEC. Ce type de caractère est utilisé pour *quoter* n'importe quel autre caractère. *Quoter* un caractère consiste à lui donner implicitement le type 8 (le type des caractères normaux). Par défaut il n'existe qu'un seul caractère de ce type, le caractère *slash* / .

2 : type BCOM. Ce type de caractère sert à indiquer le début d'un commentaire qui sera terminé à l'occurrence d'un caractère du type suivant. Par défaut il n'existe qu'un seul caractère de ce type, le caractère point virgule ; .

3 : type ECOM. Ce type de caractère sert à indiquer la fin d'un commentaire. Par défaut il n'existe qu'un seul caractère de ce type, le caractère Return

4 : type SEP. Définit un caractère séparateur standard (Ex: l'espace, la tabulation ...).

5 : type MACH. Ce type indique que le caractère est un macro-caractère. Une fonction est associée à l'atome dont le P-NAME est ce caractère. Cette fonction est invoquée automatiquement à la lecture de ce caractère dans le flux d'entrée (voir la section suivante).

6 : type CSYMB. Ce type de caractère indique le délimiteur de symboles spéciaux. Par défaut il n'y a qu'un seul cacatère de ce type, le caractère valeur absolue.

7 : type SHARP. Définit le caractère de #-macro.

8 : type REGULAR. Définit un caractère normal pouvant être utilisé pour construire un P-NAME.

9 : type DOT. Ce type de caractère (le point . par défaut) sert à écrire les paires pointées.

10 : type LPAR. Ce type de caractère (la parenthèse ouvrante (par défaut) sert de caractère de début de liste.

11 : type RPAR. Ce type de caractère (la parenthèse fermante) par défaut) sert de caractère de fin de liste.

12 : type MONOSYM. Ce type de caractère sert à définir des caractères qui sont toujours considérés comme des symboles mono-caractères.

(TYPECH <ch> <n>) **[SUBR à 1 ou 2 arguments]**

permet de connaître (si <n> n'est pas fourni) ou de modifier (si <n> est fourni) le type du caractère <ch>. Cette fonction retourne le type courant du caractère <ch> après modification éventuelle.

```
ex :  ; apres la redefinition syntaxique des
      ; caracteres < et >,

         (TYPECH '< 10)   ->  10
         (TYPECH '> 11)   ->  11

      ; l'entree :

         <CONS '(A> '<B)>

      ; est lue :
```

```
(CONS '(A) '(B))
```

(TYPECN <cn> <n>) **[SUBR à 1 ou 2 arguments]**

est identique à la fonction précédente, mais le caractère est spécifié sous la forme de son code ASCII.

4.2.5 Les macro-caractères

Un macro-caractère est un caractère quelconque auquel a été associé une fonction qui est lancée automatiquement à la lecture de ce caractère dans le flux d'entrée. La valeur retournée par cette fonction remplace le macro-caractère lu. Tout caractère peut être utilisé comme macro-caractère.

(DMC <ch> <l> <s1> ... <sN>) **[FSUBR]**

l'argument <ch> doit être un symbole mono-caractère. DMC associe à ce caractère une fonction qui possède une liste de variables locales <l> (cette liste est obligatoire à cette position même s'il n'y a pas de variables locales) et un corps de fonction <s1> ... <sN>. DMC possède la même syntaxe que les autres fonctions de définition (DE, DF et DM) et retourne <ch> en valeur.

DMC peut être défini en Lisp de la manière suivante :

```
(DF DMC L
    (APPLY 'DE L)              ; fabrique la fonction associee
    (TYPECH (CAR L) 5))        ; force le nouveau type
    (CAR L))                   ; retourne en valeur le caractere.
```

Pour détruire une définition de macro-caractère, il faut changer le type du caractère et détruire la fonction qui lui était associée par exemple au moyen de la fonction suivante :

```
(DE REMACH (ch)          ; [pour  REMove MAcro CHaracter]
    (TYPECH ch 8)        ; le caractere redevient normal
    (REMFN ch)           ; destruction de la fonction
    ch))
```

Enfin il est possible de définir des macro-caractères dynamiques en liant le type du caractère et la fonction au moyen de la fonction BIND :

```
(BIND ((TYPECH <ch> 5)
       (TYPEFN <ch> 'EXPR)
       (VALFN  <ch> <l>))
       ...
)
```

ATTENTION : la fonction associée à un macro-caractère étant de la même nature que les fonctions définies par l'utilisateur (de type DE) il n'est pas possible, pour un atome mono-caractère, de posséder tout à la fois une définition de type DE et une définition de type DMC.

Il existe 6 macro-caractères prédéfinis :

- le macro-caractère quote '
- le macro-caractère load %
- le macro-caractère flexe ^
- le macro-caractère edit &
- le macro-caractère crochet [ ]
- le macro-caractère dièse #

4.2.5.1 le macro caractère quote

Le quote (apostrophe) ' est le plus connu et le plus utilisé des macro-caractères. Placé devant une S-expression quelconque, il retourne la liste (QUOTE S-expression).

```
ex :  '(A B)    est lu comme      (QUOTE (A B))
      ''A               "         (QUOTE (QUOTE A))
```

4.2.5.2 le macro caractère load

placé devant un symbole, fabrique l'appel : (load <symbole>). Il permet donc de charger un fichier Lisp interactivement d'une manière extrêmement concise. De plus % imprime sur le flux de sortie courant le nom du fichier en cours de chargement et un point à chaque évaluation.

% peut être défini en Lisp de la manière suivante :

```
(DMC |%| () (LIST 'LOAD (READ)))

ex : %foo     est lu comme     (LOAD FOO)
```

4.2.5.3 le macro caractère flexe

ce macro caractère permet de lire un symbole mono-caractère dont le P-NAME est un caractère contrôle.

^ peut être défini en Lisp de la manière suivante :

```
(DMC |^ | () (IMPLODE (- (READCN) 32)))
```

ex : ^D est lu comme un symbole mono caractère

4.2.5.4 le macro caractère edit

placé devant un symbole, fabrique l'appel : (edit <symbole>). Il permet donc d'appeler l'éditeur sur un fichier Lisp interactivement d'une manière extrêmement concise.

& peut être défini en Lisp de la manière suivante :

```
(DMC |&| () (LIST 'PEPE (READ)))
```

```
ex : & foo    est lu comme    (PEPE foo)
```

4.2.5.5 le macro caractère crochet

permet de contruire une expression Lisp qui, évaluée, construira une autre expression Lisp.

l'expression	est lue comme :
[a b c d e]	(LIST A B C D E)
[a !b]	(CONS A B)
[a b c !d]	(MCONS A B C D)
[a !b !c !d]	(CONS A (APPEND B C D))
[!a b !c]	(APPEND A (CONS B C))
[!a !b c]	(APPEND A B (LIST C))

Ces macro-caractères peuvent être définis en Lisp de la manière suivante :

;----- une variable globale d'état

```
(setq $in-bracketp$ ())
```

```
;----- Les macros caractères eux-mêmes

(dmc |!| ()
     (if $in-bracketp$
         '|!|
         (syserror '|!| 'ERSXT ())))

(dmc |]| ()
     (if $in-bracketp$
         '|]|
         (syserror '|]| 'ERSXT ())))

(dmc |[| ()
     (let ((begin (list 'LIST)) ($in-bracketp$ T))
          (readbracket begin begin (read))
          begin)))

;-----    Lecture d'un![

(de readbracket (begin current new)
    (cond ((eq new '|]|))
          ((neq new '|!|)
           (readbracket begin (placdl current new) (read)))
          (t (let ((elemseq (list (read))))
               (if (member elemseq '((|!|) (|]|)))
                   (syserror '|!| 'ERSXT elemseq)
                   (if (eq begin current)
                       (sequence
                          (cdr (rplac begin 'APPEND elemseq))
                          (read))
                       (rplaca begin
                              (if (eq current (cdr begin))
                                  'CONS
                                  'MCONS))
                       (let ((nextseq (read)))
                            (if (eq nextseq '|]|)
                                (rplacd current elemseq)
                                (sequence
                                  (cdadr
                                     (rplacd current
                                             (list (cons 'APPEND
                                                         elemseq))))
                                  nextseq)))))))))))

(de sequence (last next)
    (if (eq next '|!|)
        (sequence (placdl last (read)) (read))
        (when (neq next '|]|)
              (let ((begin (list 'LIST next)))
                   (rplacd last (list begin))
                   (readbracket begin (cdr begin) (read)))))))
```

4.2.5.6 le macro caractère dièze

ce macro caractère permet de convertir des nombres ou d'appeler l'interprète au cours d'une lecture normale. Son fonctionnement est très général et s'explique ainsi : à l'occurrence d'un caractère # dans le flux d'entrée, le lecteur Lisp lit le caractère suivant au moyen de la fonction READCH. Ce caractère, le *sélecteur de #–macro,* doit posséder sur sa P-LIST une fonction sous l'indicateur SHARP. Cette fonction est lancée et la valeur retournée remplace le #-caractère. Il est possible de définir de nouvelles #-macros au moyen de la fonction suivante.

(SHARP <ch> <lvar> <s1> ... <sN>) **[FSUBR]**

définit une nouvelle #-macro. <ch> est le nouveau caractère sélecteur, <lvar> est la liste des variables de la fonction associée à cette macro et <s1> ... <sN> est le corps de cette fonction. SHARP possède donc la même syntaxe que la fonction DMC.

SHARP peut être défini en Lisp de la manière suivante :

```
(DF SHARP (f . l)
    (PUTPROP f (CONS 'LAMBDA l) 'SHARP)))
```

Il existe un certain nombre de sélecteurs prédéfinis :

#/ retourne le code interne (ASCII) du caractère suivant lu sur le flux d'entrée. C'est le meilleur moyen connu à ce jour pour entrer des codes ASCII dans un programme. #/A remplace avantageusement 65 pour décrire le code interne du caractère A.

```
ex : #/A     est lu comme     65
     #/a         " "          97
```

#^ retourne le code interne (ASCII) du caractère suivant lu sur le flux d'entrée considéré comme un caractère contrôle, c'est-à-dire avec les bits 6 et 7 mis à 0.

```
ex : #^ C    est lu comme     3
     #^ Z        " "          21
```

#$ lit le nombre suivant en base seize.

```
ex : #$FF    est lu comme     255
     #$10        " "          16
```

#. évalue l'expression suivante du flux d'entrée. Cette valeur est retournée par la #-macro.

```
ex : #.(1+ 4)    est lu comme    5
```

#" lit une liste de caractère terminée par le caractère " et retourne la liste de leur code ASCII. Pour y insèrer le caractère " il sut ebé é é Œ97 34 98°

4.3 Les fonctions de sortie de base

Toutes éditent dans un tampon de sortie totalement accessible à l'utilisateur. Ce tampon est périodiquement vidé dans le flux de sortie qui est associé soit à un terminal soit à un fichier sélectionné au moyen de la fonction OUTPUT.
Si au cours d'une édition le tampon de sortie devient plein, il est automatiquement vidé, en utilisant la fonction (TERPRI 1), et l'édition se poursuit dans un nouveau tampon vide.
On peut, à tout moment, suspendre une impression sur le terminal *(holding)* en entrant sur le terminal le caractère spécial ^S (propre à CP/M). L'impression reprend avec la frappe du caractère ^Q.

(PRIN <s1> ... <sN>) **[SUBR à N arguments]**

édite dans le tampon de sortie la valeur des différentes S-expressions <s1> ... <sN> sans vider le tampon. PRIN retourne en valeur la valeur de son dernier argument c'est-à-dire <sN>.

(TERPRI <n>) **[SUBR à 0 ou 1 argument]**

vide dans le flux de sortie tous les caractères du tampon de sortie (comme pour la fonction précédente), efface ce tampon, puis saute <n> lignes en imprimant le code Return suivi de <n> fois le code Line-feed. Si l'argument <n> n'est pas fourni, TERPRI agit comme si <n> était égal à 1 provoquant une impression avec interlignage simple. L'appel (TERPRI) est donc équivalent à (TERPRI 1).

(FLUSH) **[SUBR à 0 argument]**

vide dans le flux de sortie tous les caractères du tampon de sortie et efface celui-ci. Aucun autre caractère n'est envoyé dans le flux de sortie.

(PRINT <s1> ... <sN>) **[SUBR à N arguments]**

édite dans le tampon de sortie les différentes S-expressions <s1> ... <sN> puis vide ce tampon (en faisant un appel implicite à la fonction (TERPRI 1)). PRINT retourne en valeur la valeur de son dernier argument c'est-à-dire celle de <sN>.

PRINT peut être défini en Lisp de la manière suivante :

```
(DE PRINT s
    (PROG1 (APPLY 'PRIN s) (TERPRI 1)))

ex : ? (PRINT (1+ 9) (CDR '(A B C)))    ; forme a evaluer
       10(B C)                          ; execution
     = (B C)                            ; valeur retournee
```

(PRINCH <ch> <n>) **[SUBR à 1 ou 2 arguments]**

édite <n> fois le caractère <ch> dans le tampon de sortie. Si <n> n'est pas un nombre ou est omis, le caractère <ch> n'est édité qu'une fois. PRINCH retourne <ch> en valeur.

```
ex : (PRINCH '|+| 10)  ; edite  10 caracteres + dans le tampon

     (DE PYR (n1 n2)
         (WHEN (>= n1 0)
               (PRINCH '| | n1)
               (PRINCH '|*| (1+ (* n2 2)))
               (TERPRI)
               (PYR (1- n1) (1+ n2)))

     (PYR 4 0)

     produira

        *
       ***
      *****
     *******
```

(PRINCN <cn> <n>) **[SUBR à 1 ou 2 arguments]**

comme la fonction précédente, édite <n> fois le code ASCII <cn> dans le flux de sortie. Si <n> est omis, le code <cn> n'est édité qu'une fois. PRINCN retourne le code ASCII <cn> en valeur.

```
ex :  (LET ((cn #/A))
           (REPEAT 10 (PRINCN (INCR cn))))

      ; produira

      BCDEFGHIJK
```

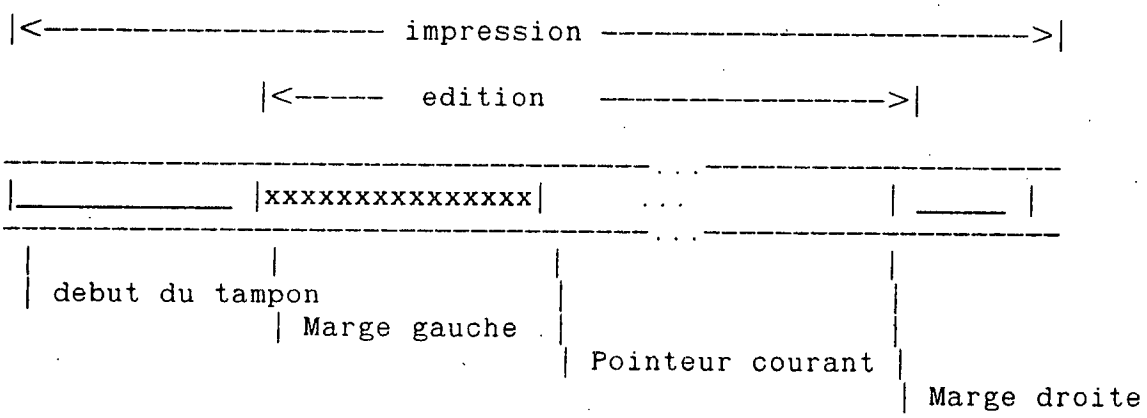
4.4 Le contrôle des fonctions de sortie

Les fonctions de sortie éditent les représentations externes des objets Le_Lisp dans un tampon de sortie accessible au moyen de fonctions spécialisées.

La plupart de ces fonctions vont permettre de contrôler un paramètre interne des fonctions de sortie et possèdent 0 ou 1 argument. Un appel sans argument est une demande de la valeur du paramètre *(GET)*, tandis qu'un appel avec 1 argument provoque un changement du paramètre *(SET)*. Le_Lisp utilise des fonctions (et non des variables) pour contrôler les paramètres internes car elles réalisent des tests de validité. En cas de changement temporaire d'un de ces paramètres il est commode de s'assurer de la restitution du paramètre modifié au moyen de la fonction PROTECT ou BIND.

4.4.1 Le tampon de sortie

Le tampon de sortie est organisé de la manière suivante :



Les éditions ne s'effectuent dans le tampon de sortie qu'entre la marge gauche et la marge droite alors que les impressions vont porter sur tout le tampon. L'accès aux valeurs de la marge gauche, du pointeur courant et de la marge droite est réalisé au moyen de fonctions spéciales ; il existe de plus une fonction OUTBUF permettant de manipuler directement n'importe quel caractère de ce tampon de sortie. Les fonctions qui vont suivre testent la validité de leur argument et provoquent l'erreur EROOB en cas d'argument hors limite.

(LMARGIN <n>) **[SUBR à 0 ou 1 argument]**

permet de spécifier la marge à gauche <n> de l'impression. Par défaut à l'initialisation du système on a (LMARGIN 0). Si <n> n'est pas fourni, la marge n'est pas modifiée. LMARGIN retourne la valeur de la marge gauche courante et est principalement utilisée par le Pretty-Print (voir le chapitre sur le Pretty-Print) pour gérer automatiquement les renfoncements gauches à l'impression des structures de contrôle.

(RMARGIN <n>) **[SUBR à 0 ou 1 argument]**

permet de spécifier la marge à droite <n> de l'impression. Par défaut à l'initialisation du système on a (RMARGIN 72). Si <n> n'est pas fourni, la marge n'est pas modifiée. RMARGIN retourne la valeur de la marge droite courante et est utilisée principalement pour régler la taille des lignes en fonction du terminal de sortie utilisé (télétype, écran, imprimante ...).

(OUTPOS <n>) **[SUBR à 0 ou 1 argument]**

permet de spécifier, si <n> est fourni la nouvelle valeur du pointeur courant sur le tampon de sortie. Cet index pointe toujours sur la 1ère position libre dans le tampon. OUTPOS retourne en valeur la position courante de ce pointeur.

(OUTBUF <n> <cn>) **[SUBR à 1 ou 2 arguments]**

considère le tampon de sortie comme un vecteur et permet donc d'avoir accès et/ou de modifier le <n>ième caractère du tampon de sortie avec le code interne <cn>, si l'argument <cn> est donné. OUTBUF retourne toujours la valeur du <n>ième caractère du tampon de sortie.

4.4.2 Les limitations d'impression

Pour pouvoir limiter les impressions des structures très longues ou partagées et éviter de faire boucler l'impression en cas de listes circulaires trois nouvelles fonctions ont été introduites.

(PRINTLENGTH <n>) **[SUBR à 0 ou 1 argument]**

permet de connaître et/ou de modifier (si l'argument numérique <n> est fourni) le nombre maximum d'éléments de liste qui est imprimé lors d'un PRINT ou d'un PRIN. Par défaut la valeur de ce nombre est de 2000. Si une liste contient un nombre d'éléments plus élevé, le reste de ses éléments ne sera pas imprimé et des points de suspension ... apparaîtront devant la parenthèse fermante. Cette fonction permet de terminer l'impression des listes circulaires sur les CDR. PRINTLENGTH retourne en valeur la longueur maximum courante d'impression.

```
ex : (PRINTLENGTH 6)        -> 6
     '(1 2 3 4 5 6 7 8 9) -> (1 2 3 4 5 6...)
```

(PRINTLEVEL <n>) **[SUBR à 0 ou 1 argument]**

permet de connaître et/ou de modifier (si l'argument numérique <n> est fourni) la profondeur maximum d'impression lors d'un PRINT ou d'un PRIN. Cette valeur représente le nombre maximum de parenthèses ouvrantes non encore refermées. En cas de dépassement, la liste qui provoque le débordement n'est pas imprimée et le caractère & est imprimé à sa place. Cette fonction permet d'imprimer des listes circulaires sur les CAR. PRINTLEVEL retourne en valeur la profondeur maximum courante d'impression.

```
ex : (PRINTLEVEL 3) -> 3

     '(DE FOO (1)
          (IF (NULL (CDR 1))
              1
              (FOO (CDR 1))))
     ->  (DE FOO (1) (IF (NULL &) 1 (FOO &)))

; impression des listes circulaires ;

?  (PRINTLEVEL 10)
=  10
?  (PRINTLENGTH 50)
=  50
?  (SETQ L '(X Y Z))
=  (X Y Z)
?  (RPLACD (CDDR L) L)
=  (Z X Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X
    Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X Y Z X...)
?  (RPLACA L L)
```

```
= (((((((((((& Y Z & Y Z & Y Z & Y Z & Y Z & Y Z & Y Z &
  YZ & Y Z & Y Z & Y Z & Y Z & Y Z & Y Z & Y Z & Y Z &
  Y..)...)...)...)...)...)...)...)...)...)
?
```

(PRINTLINE <n>) **[SUBR à 0 ou 1 argument]**

permet de connaître et/ou de modifier (si l'argument numérique <n> est fourni) le nombre maximum de lignes (c'est-à-dire d'appels à la fonction TERPRI) qui vont être imprimées lors d'un PRINT ou un PRIN. Par défaut la valeur de ce nombre est 5000. Si une expression nécessite plus de lignes, la dernière ligne apparaîtra avec des points de suspension. PRINTLINE retourne en valeur le nombre maximum de lignes autorisé pour une édition.

4.4.3 L'impression des listes circulaires

L'utilisation des fonctions de limitation d'impression du paragraphe précédent, pour autant qu'elles stoppent l'impression et permettent d'en avoir une vue abrégée, ne permettent pas, en cas de structures circulaires ou partagées de connaître les cellules effectivement partagées.

Pour cela il faut utiliser les fonctions suivantes, écrites en Lisp qui utilisent la propriété de la fonction EQ (ici de la fonction MEMQ) suivante : (EQ l1 l2) est vrai si les 2 listes l1 et l2 sont les mêmes physiquement c'est-à-dire ont la même adresse physique.

```
; Fonctions d'impressions circulaires simples :

(DE CPRINT (l)
    ; Fonction d'impression de l
    (CPRIN l)
    (TERPRI)
    T)

(DE CPRIN (l)
    ; Fonction d'edition de l
    (LET ((vus))
         ; vus contient la liste des objets deja visites
         (CPRIN1 l)))

(DE CPRIN1 (l)
    ; Fonction auxilaire
    (IF (ATOM l)
        ; Les atomes peuvent etre partages
        (PRIN l)
        ; En cas de liste
        (PRINCH '|(|)
```

```
        (WHILE (CONSP l)
               ; iteration sur les CDR
               (IF (MEMQ l vus)
                   ; c'est un objet deja visite
                   (SETQ l '|...|)
                   ; c'est un objet nouveau
                   (NEWL vus l)
                   ; recurse sur les CAR
                   (CPRIN1 (NEXTL l))
                   ; separation inter element d'une liste
                   (IF l (PRINCH '| |)))))
       (IF (NULL l)
           ; La liste se termine bien
           ()
           ; C'est une paire pointee
           (PRIN '|. | l))
       ; Fermante finale
       (PRINCH '|)|)))))

; ainsi si    (SETQ LL '(A B C D))

; l'appel     (CPRINT (NCONC LL LL))

; produira    (A B C D . ...)
```

4.4.4 L'édition standard

Les différents objets Lisp sont édités dans le tampon de sortie en respectant l'unique règle suivante : la représentation externe d'un objet atomique ne peut etre éditée à cheval sur deux lignes; cela pour les symboles et les nombres.

La fonction SPRINT permet de régler certaines modalités d'impression, la fonction OBASE règle la base de conversion des nombres et la fonction PTYPE permet de manipuler aisément le mode d'impression de chaque symbole (le P-type).

(SPRINT <i>) **[SUBR à 0 ou 1 argument]**

cette fonction permet de connaître ou de modifier l'indicateur d'impression des P-NAME spéciaux. Si <i> = T, les symboles spéciaux sont encadrés du caractère | ce qui permet de faire des impressions relisibles par la fonction Lisp READ.

(OBASE <n>) **[SUBR à 0 ou 1 argument]**

permet de modifier (si l'argument <n> est fourni) ou de consulter la valeur de la base de conversion des nombres en sortie. Seule la base 16 est disponible actuellement.

(PTYPE <sym> <n>) **[SUBR à 1 ou 2 arguments]**

permet de modifier (si l'argument <n> est fourni) ou de consulter (si l'argument <n> n'est pas fourni) la valeur du P-TYPE du symbole <sym>. Ce P-TYPE est principalement utilisé par le PRETTY-PRINT pour y stocker le format à utiliser pour éditer cet atome symbole en tant que fonction. PTYPE retourne la valeur courante du P-TYPE du symbole <sym>.

4.4.5 L'interruption fin de ligne

Quand le système décide de vider le tampon de sortie, implicitement si un atome ne rentre pas dans le tampon, ou explicitement par appel de la fonction TERPRI, il déclenche une interruption programmée de nom EOL. En temps normal, cette fonction vide le tampon de sortie dans le flux de sortie courant et efface tout le tampon mais il est parfois souhaitable de prendre le contrôle de cette fonction pour copier le tampon sur plusieurs flux, le recopier dans la mémoire, transformer certains caractères du tampon ... Toutes ces actions vont être réalisables en redéfinissant temporairement la fonction EOL.

(EOL) **[SUBR à 0 argument]**

vide le tampon de sortie dans le flux de sortie courant. Cette fonction est appelée automatiquement par le système et n'existe en tant que fonction que pour être redéfinie. Ces redéfinitions vont permettre de rediriger le flux de sortie (voir la fonction suivante).

(STREAM-OUTPUT <s1> ... <sN>) **[FSUBR]**

permet de récupérer sous forme de liste de listes de caractères toutes les impressions produitent durant le temps de l'évaluation des expressions <s1> ... <sN>. Chaque tampon vidé produit une liste de caractères et STREAM-OUTPUT retourne une liste de ces lignes. Les tampons ne sont pas copiés dans le flux de sortie. Cette fonction est extrêmement utile en particulier pour l'éditeur PEPE qui peut ainsi récupérer tout ce qui peut être imprimé en vue de son édition.

STREAM-OUTPUT peut être défini en Lisp de la manière suivante :

```
(DF STREAM-OUTPUT 1
    (LET ((ll) (i) (r))
         (FLET ((EOL ()
```

```
        (SETQ i 0 r ())
        ; les caracteres dans r
        (REPEAT (OUTPOS)
                (NEWL r (OUTBUF i))
                (OUTBUF i #/ )
                (INCR i))
        ; reconstruit la marge gauche
        (OUTPOS 0)
        (REPEAT (LMARGIN)
                (OUTBUF (OUTPOS) #/ )
                (OUTPOS (1+ (OUTPOS))))
        ; fabrique la liste des lignes
        (NEWR ll r)))
    (EPROGN 1)
    ll))))
```

4.5 Les fonctions sur les flux d'entrée/sortie

Dans l'état minimum du système, Le_Lisp 80 peut gérer simultanément
- un terminal d'entrée/sortie
- un fichier d'entrée
- un fichier de sortie
- un fichier d'entrée auxiliaire.

Toutes les fonctions de cette section vont utiliser le système de gestion de fichiers du système hôte (ici CP/M) qui peut retourner un code erreur en cas d'anomalie. Si c'est le cas, l'erreur ERIOS se déclenche dont le libellé est :

```
** <fn> : erreur disque
```

dans lequel <fn> est le nom de la fonction Lisp qui était appelée.

Une spécification de fichier <file> est en Lisp un symbole. La fonction suivante permet de convertir un symbole en une vraie spécification de fichier propre au système hôte (ici CP/M).

(FILENAME <sym1> <sym2>) **[SUBR à 1 ou 2 arguments]**

<sym1> doit être un symbole représentant un nom de fichier. FILENAME rajoute l'extension <sym2> (ou bien LL si <sym2> n'est pas fourni) à ce nom dans un format propre à CP/M.

```
ex :  (FILENAME 'foo)          ->  |FOO      LL |
      (FILENAME 'bar 'lg)      ->  |BAR      LG |
```

; voici le bon moyen d'ouvrir un fichier :

```
(OR (INPUT (FILENAME file))
    (SYSERROR 'FILENAME '|quezako| file))
```

(SELECTDRIVE <sym>) **[SUBR à 1 argument]**

permet, sous CP/M, de changer d'unité de disque. <sym> doit être une lettre comprise entre A et P.

SELECTDRIVE peut être défini en Lisp de la manière suivante :

```
(DE SELECTDRIVE (sym)
    (EXECUTE
       [#$1E (- (cascii sym) #/A)      ; MVI E, 0/1/2 ...
       #$0E 14                          ; MVI C, code
       #$CD 05 00])                     ; CALL 0005
    sym)

ex : (SELECTDRIVE 'D)   ->   D
```

4.5.1 La sélection des flux d'entrée/sortie

(INPUT <flux>) **[SUBR à 1 argument]**

cette fonction permet de sélectionner le flux d'entrée qui va être utilisé pour toutes les opérations de lecture suivantes. Ce flux peut être :
NIL, il s'agit alors du terminal.
T, il s'agit alors du périphérique auxiliaire.
<file>, il s'agit alors d'un fichier sur disque.

INPUT retourne T si le fichier est correctement ouvert et NIL dans le cas contraire.

(EOF) **[SUBR à 0 argument]**

cette fonction est automatiquement lancée par le système à la fin du fichier d'entrée. EOF peut être également lancée par l'utilisateur pour provoquer artificiellement des fins de fichiers en lecture (le "SIGNAL EOF;" cher aux PL1istes). Par défaut EOF ferme le fichier et provoque un échappement de nom EOF.
Il n'y a jamais de fin de fichier sur le terminal en entrée sauf à définir un macro-caractère de fin de fichier.

EOF peut être défini en Lisp de la manière suivante :

```
(DE EOF ()
    (INPUT ())
    (EXIT EOF 'EOF))
```

ex : (DMC ^ Z () '(EOF))

(OUTPUT <flux>) [SUBR à 1 argument]

cette fonction permet de sélectionner le flux de sortie dans lequel vont être dirigées toutes les sorties suivantes. Ce flux peut être :
NIL, il s'agit alors du terminal.
T, il s'agit alors du périphérique auxiliaire.
<file>, il s'agit alors d'un fichier sur disque.

OUTPUT retourne T si le fichier a été convenablement ouvert et NIL dans le cas contraire.

4.5.2 La fonction LOAD et le mode AUTOLOAD

(LOAD <file>) [FSUBR]

permet de charger (en évaluant toutes les expressions qui s'y trouvent) le fichier de nom <file>. LOAD retourne T en valeur.

(LOADFILE <file>) [SUBR à 1 argument]

est identique à la fonction précédente mais le nom du fichier <file> est évalué. LOADFILE retourne toujours T en valeur.

(AUTOLOAD <file> <sym1> ... <symN>) [FSUBR]

Comme il est fastidieux de charger à la main plusieurs fichiers de fonctions à chaque exécution, Le_Lisp permet l'utilisation de fonctions *autoload* (qui se chargent toutes seules dynamiquement à leur premier appel).

<file> est le nom d'un fichier qui doit être automatiquement chargé, au moyen de la fonction précédente, si un des symboles <sym1> ... <symN> doit être évalué en tant que fonction. Le chargement du fichier suit la convention des appels par nécessité.

AUTOLOAD peut être défini en Lisp de la manière suivante :

```
(DF AUTOLOAD (f . l)
    ; (AUTOLOAD fichier atl ... atN)
    (MAPC (LAMBDA (at)
              (EVAL (LIST 'DM at 'l
                          (LIST 'REMFN (KWOTE at))
                          (LIST 'LOADFILE f)
                          'l)))
          l)))
```

Ainsi l'appel :

```
(AUTOLOAD PRETTY PRETTY)
```

donne la définition suivante à la fonction PRETTY :

```
(DM PRETTY l
    (REMFN 'PRETTY)
    (LOADFILE 'PRETTY)
    l)
```

ce qui fait qu'à la première évaluation de (PRETTY ...) la MACRO est appelée. Cette MACRO se détruit elle-même (pour éviter de boucler au cas où la fonction ne serait pas définie) puis évalue (LOADFILE 'PRETTY) qui charge en silence le fichier PRETTY
La valeur retournée par la MACRO étant l'appel originel (PRETTY ...) lui-même, EVAL reévalue cette forme dans laquelle la fonction PRETTY est maintenant définie (cette nouvelle définition se trouvait dans le fichier). Ouf...

CHAPITRE 5

LA GESTION DU TERMINAL

Un ensemble de fonctions permettent de travailler directement sur le terminal (TTY) et d'utiliser au mieux toutes ses possibilités. Ces fonctions n'utilisent pas le système d'entrées-sorties classique et peuvent donc être utilisées à tout moment. De plus elles sont indépendantes du type du terminal.

5.1 Les fonctions de base

(TYI) [SUBR à 0 argument]

retourne un nombre représentant le code interne du caractère suivant lu sur le clavier du terminal, sans aucune conversion de casse et sans utiliser l'éditeur de ligne du moniteur (en particulier les caractères RUB-OUT, et les différents caractères de contrôle ne sont plus effectifs).

(TYS) [SUBR à 0 argument]

si un caractère est prêt à être lu sur le clavier du terminal, retourne la valeur que retournerait la fonction TYI, sinon TYS retourne NIL. Cette fonction permet de tester si un caractère a été frappé sur le terminal. Si un caractère est effectivement retourné il est réellement lu.
Donc (UNTIL (TYS)) est bien équivalent à (TYI).

ATTENTION : quand un programme Lisp travaille, tout appui sur une touche du clavier provoque l'appel de la fonction ITEVAL (voir le chapitre sur les fonctions système). Une utilisation interactive du système amène souvent à redéfinir cette fonction pour éviter de perdre des caractères.

(TYO <cn1> ... <cnN>) [SUBR à N arguments]

édite dans le tampon du terminal tous les caractères dont les codes internes sont <cn1> ... <cnN>. Si les arguments sont des listes, TYO les considèrent comme des listes de codes internes.
Il n'y pas de conversion de caractère au moment de la sortie, en particulier les caractères contrôles sont envoyés directement sans passer par la forme ^X. Il est donc possible d'envoyer au moyen de cette fonction les caractères de contrôle (comme le déplacement du curseur) propre à chaque terminal.

Si le code à imprimer est plus grand que 128, TYO considère que les bits de poids forts représentent un *facteur de répétition* de ce caractère . Il est donc possible de représenter <n> caractères <c> identiques sous la forme d'un seul nombre (+ (* (1- <n>) 128) <c>).

```
(TYO #/A #/B)                imprime les caracteres
     AB
(TYO '#"Soft Machine")       imprime les caracteres
     Soft Machine
(TYO #.(+ (* 4 128) #/*)) imprime les caracteres
     *****
```

5.2 Les fonctions du terminal virtuel VIRTTY

Les variables et les fonctions qui vont suivre sont dépendantes du terminal (donc du système) utilisé. Elles sont initialisées et/ou définies dans un fichier système, de nom VIRTTY qui est toujours chargé au lancement du système. La description d'un nouveau terminal doit comporter au minimum l'initialisation des variables globales $XMAX$, $YMAX$, $LEFT$, $RIGHT$, UP, $DOWN$ et la définition effective des deux fonctions TYCLS et TYCURSOR. Toutes les autres fonctions doivent également être présentes mais peuvent retourner la valeur NIL dans le cas où elles ne seraient pas effectives du fait des limitations du terminal. La lecture du fichier VIRTTY montre diverses descriptions de terminaux.

$XMAX$ [Variable]

co tient l'indice maximum des colonnes du terminal compté à partir de 0. En général $XMAX$ vaut 79 (pour les terminaux qui ont 80 colonnes).

$YMAX$ [Variable]

contient l'indice maximum des lignes du terminal compté à partir de 0. En général $YMAX$ vaut 23 (pour les terminaux qui ont 24 lignes).

$LEFT$ [Variable]

contient le code clavier associé à la touche *flèche à gauche*.

$RIGHT$ [Variable]

contient le code clavier associé à la touche *flèche à droite*

UP [Variable]

contient le code clavier associé à la touche *flèche en haut*

$DOWN$ [Variable]

contient le code clavier associé à la touche *flèche en bas*

(TYCLS) [SUBR à 0 argument]

çù.efface tout l'écran. Retourne T si c'est possible et NIL dans le cas contraire.

(TYCURSOR <x> <y>) [SUBR à 2 arguments]

positionne le curseur en colonne <x> à la ligne <y>. La position en haut à gauche de l'écran correspond aux indices 0,0. Retourne T si c'est possible et NIL dans le cas contraire.

(TYCO <x> <y> <cn1> ... <cnN>) [SUBR à N arguments]

correspond à l'appel :

```
(PROGN (TYCURSOR x y) (TYO cn1 ... cnN))
```

c'est-à-dire se positionne à la position <x> <y> puis édite dans le tampon du terminal les codes cn1 ... cnN comme pour la fonction TYO.

(TYBEEP) [SUBR à 0 argument]

déclenche l'alarme (clochette, buzzer, ...) sur le terminal. Retourne T si c'est possible et NIL dans le cas contraire.

(TYCLEOL) [SUBR à 0 argument]

efface l'écran de la position du curseur à la fin de la ligne. Retourne T si c'est possible et NIL dans le cas contraire.

(TYCLEOS) [SUBR à 0 argument]

efface l'écran de la position du curseur à la fin de l'écran. Retourne T si c'est possible et NIL dans le cas contraire.

(TYINSCH <cn>) [SUBR à 1 argument]

insère le caractère de code ASCII <cn> à la position courante du curseur. Retourne T si c'est possible et NIL dans le cas contraire.

(TYDELCH) [SUBR à 0 argument]

détruit le caractère à la position courante du curseur. Retourne T si c'est possible et NIL dans le cas contraire.

(TYINSLN) [SUBR à 0 argument]

insère une nouvelle ligne à la position courante du curseur. Retourne T si c'est possible et NIL dans le cas contraire.

(TYDELLN) [SUBR à 0 argument]

détruit la ligne à la position courante du curseur. Retourne T si c'est possible et NIL dans le cas contraire.

(TYATTRIB <i>) [SUBR à 1 argument]

si l'indicateur <i> est vrai, active le mode attribut, s'il est faux, l'enlève. Le mode attribut qui sert à mettre en valeur un texte sur l'écran dépend du terminal : il peut s'agir du soulignement, du clignotement (attention au mal de crâne) ou de l'affichage inversé. Retourne T si c'est possible, NIL dans le cas contraire.

5.3 Utilisation du terminal virtuel

Il existe 2 programmes de démonstration du terminal virtuel :
- HANOI qui simule le jeu du même nom
- VDT qui simule la dure vie d'un ver-de-terre.

Ils se trouvent respectivement dans les fichiers HANOI et VDT Leur lecture est facile et permet de bien comprendre l'utilisation du terminal simple, du terminal virtuel et des interruptions de EVAL.

(HANOI <n>) [EXPR à 1 argument]

anime le jeu des tours de Hanoi avec <n> disques. <n> doit être un nombre compris entre 3 et 9. Cette fonction est chargée automatiquement à son premier appel.

(HANOIEND) [SUBR à 0 argument]

permet de récupérer la place occupée par les fonctions du jeu précédent.

(VDT) [EXPR à 0 argument]

lance un jeu d'animation qui simule le croissance d'un ver de terre. Utilisez les flèches de votre terminal pour contrôler le mouvement du ver. Cette fonction est chargée automatiquement à son premier appel.
It's more fun to compete

(VDTEND) [EXPR à 0 argument]

permet, quand le travail sérieux reprend, de récupérer l'espace occupé par les fonctions du jeu précédent.

CHAPITRE 6

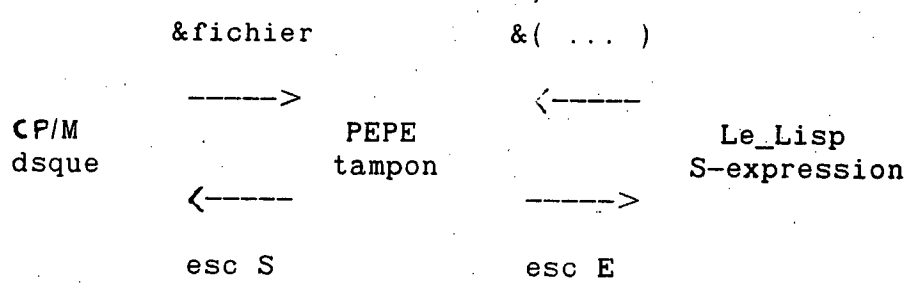
L'EDITEUR INTEGRE PEPE

L'une des applications immédiates d'un terminal vidéo virtuel est la réalisation d'un éditeur vidéo (appelé également éditeur *pleine page* (comme PEPE)). Cela est facilement réalisé en Lisp. La lecture des fichiers qui composent cet éditeur vous convaincra très certainement.
L'utilisation de Lisp comme langage d'implantation d'un tel éditeur permet, en plus de la facilité d'écriture et de correction, de l'étendre très rapidement.

PEPE est contenu dans les fichiers suivants :

PEPE	l'initialisateur
PEPTOP	la boucle principale
PEPLN2	la gestion des lignes
PEPMSC	les fonctions auxiliaires
PEPCMD	les commandes simples
PEPESC	les commandes ESC
PEPINI	les affectations initiales
PEPATO	les fonctions autoload

PEPE peut être vu comme un interface entre le disque et le monde Lisp :



6.1 Les Fonctions de PEPE

(PEPE <f>) [FEXPR]

appelle l'éditeur sur l'objet <f> non évalué. Il existe une autre écriture plus facile pour appeler l'éditeur, le macro-caractère & :

```
&f      correspond a l'appel  (PEPE f)
```

Si l'objet <f> est (), l'édition reprend sur le même tampon. Si l'objet <f> est T, l'édition commence avec un nouveau tampon de nom TMP. Si l'objet est un symbole, l'édition a lieu sur le fichier de même nom. Si l'objet est une liste, elle est évaluée et l'édition s'effectue dans un tampon de nom TMP qui contient tout ce qui a été imprimé par l'évaluation précédente.

; pour éditer un fichier disque de nom FOO.LL

```
&foo
```

; pour éditer la représentation paragraphée
; des fonctions FOO et BAR

```
&(pretty foo bar)
```

; pour éditer la liste triée des fonctions du système

```
&(mapc 'print
       (SORTL (MAPCOBLIST (LAMBDA (x)
                          (WHEN (TYPEFN x) [x])))))
```

(PEPFILE <f>) [EXPR à 1 argument]

permet d'appeler l'éditeur sur le fichier <f>. Cette fonction contrairement à la précédente évalue son argument.

(PEPEND) [EXPR à 0 argument]

permet de récupérer l'espace occupé par les fonctions de l'éditeur. Tout appel postérieur de PEPE rechargera l'ensemble de ces fonctions. De plus le tampon est également perdu.

6.2 Les Commandes de PEPE

^A		va au début de la ligne
^B	←	recule d'un caractère
^C		sort temporairement de PEPE, retour par &()
^D		détruit le caractère courant
^E		va en fin de ligne
^F	→	avance d'un caractère
^L		reaffiche tout l'écran
^M	RC	change de ligne à la position du curseur
^N	↓	passe à la ligne suivante
^O		casse la ligne à la position du curseur
^P	↑	passe à la ligne précédente
^T		détruit la ligne courante
^V		passe à l'écran suivant
DEL		détruit le caractère à gauche du curseur
esc E		exécute le tampon courant
esc F		change le nom du fichier courant
esc I		insère un autre fichier
esc R		lit un nouveau fichier
esc S		sauve le tampon courant
esc V		passe à l'écran suivant
esc W		écrit le tampon
esc Z		sauve le tampon et le charge en mémoire
esc)		se positionne sur la) correspondante
esc <		va au début du tampon
esc >		va en fin du tampon
esc ?		édite cette liste

CHAPITRE 7

LE PARAGRAPHEUR DE FONCTIONS

Les fonctions pré-définies PRINT et PRIN sont d'ordinaire utilisées pour éditer les S-expressions Le_Lisp. Les seules mesures prises pour améliorer la lisibilité sont :
- l'insertion d'un espace entre chaque atome ;
- l'interdiction d'éditer un atome (symbole ou nombre) à cheval sur deux lignes.

Ces mesures sont nettement insuffisantes pour éditer vos programmes. Les fonctions du paragrapheur (ou Pretty-Print en anglais) vont les éditer d'une manière beaucoup plus lisible en faisant resortir, au moyen de renfoncements gauches et de sauts de lignes ad hoc, la structure de contrôle de vos fonctions.

Pour améliorer encore la lisibilité, les appels de la fonction QUOTE sont représentés au moyen du macro-caractère ', la constante NIL par () et les macro-générations des LET sous leur forme d'entrée.

Les fonctions qui vont être décrites se trouvent dans un fichier disque de nom PRETTY et sont AUTOLOAD, c'est-à-dire qu'elles sont chargées automatiquement au premier appel de l'une d'elles.

Pour démontrer l'utilité du paragrapheur, voici une fonction telle qu'elle est imprimée par les fonctions normales :

```
? (GETDEF 'P-COND)
= (DE P-COND NIL (BIND ((LMARGIN (+ (LMARGIN) 3)))
(WHILE (CONSP L) (TERPRI) (IF (P-INLINEP (CAR L))
(P-L (NEXTL L)) (PRINCN 40) ((LAMBDA (L F) (P-P (
NEXTL L)) (WHEN L (P-PROGN T))) (NEXTL L) T)(
PRINCN 41)))))
```

Et voici cette même fonction traitée par le paragrapheur :

```
? (PRETTY P-COND)

(DE P-COND ()
    (BIND ((LMARGIN (+ (LMARGIN) 3)))
          (WHILE (CONSP L)
                 (TERPRI)
                 (IF (P-INLINEP (CAR L))
                     (P-L (NEXTL L))
                     (PRINCN 40)
```

```
(LET ((L (NEXTL L)) (F T))
     (P-P (NEXTL L))
     (WHEN L (P-PROGN T)))
(PRINCN 41)))))
```

7.1 Les fonctions du paragrapheur

(PPRINT <s>) **[EXPR à 1 argument]**

paragraphe l'expression <s> dans le flux de sortie courant. PPRINT retourne <s> en valeur.

(PRETTY <sym1> ... <symN>) **[FEXPR à N arguments]**

<sym1> ... <symN> sont des symboles qui possèdent une définition de fonction. PRETTY va éditer ces fonctions dans le flux de sortie courant et retourner NIL en valeur.

(PRETTYF <f> <sym1> ... <symN>) **[FEXPR à N arguments]**

est identique à la fonction précédente mais édite les fonctions <sym1> ... <symN> dans un fichier de nom <f>.

(PRETTYEND) **[EXPR à 0 argument]**

permet de récupérer la place occupée par les fonctions du paragrapheur après usage (de l'ordre de 500 CONS). Les fonctions du paragrapheur redeviennent AUTOLOAD.

7.2 Les formats du paragrapheur

Le paragrapheur utilise les formats suivants :

Format 1 : de type PROGN

Format 2 : de type IF

Format 3 : de type DE

Format 4 : de type COND

Format 5 : de type SELECTQ

Format 6 : de type SETQ

Ces différent formats sont rangés dans le P-type des symboles. L'accès à ce P-type est réalisé au moyen de la fonction pré-définie PTYPE.

CHAPITRE 8

LES FONCTIONS SYSTEME

8.1 L'appel et la sortie de l'interprète

(SYSTEM) **[SUBR à 0 argument]**

retourne le nom du système Le_Lisp 80. Ce nom est l'un des symboles suivants :

```
MICRAL
LX529E
SILZ
```

(VERSION) **[SUBR à 0 argument]**

retourne le numéro de version du système Le_Lisp 80 (aujourd'hui la 12).

```
ex : (VERSION)   ->   12
```

(END) **[SUBR à 0 argument]**

arrête l'évaluation en cours, ferme tous les fichiers ouverts, sort de l'interprète, ne passe pas par la case départ et rend le contrôle au moniteur. END est utilisé pour sortir définitivement de l'interprète Le_Lisp 80 et pour revenir au moniteur standard. END est la seule fonction prédéfinie qui ne retourne pas de valeur...

```
ex : ? (END)
     Que Le_Lisp soit avec vous.

     A>        retour sous CP/M
```

8.2 Le TOP–LEVEL

La boucle principale (ou TOP-LEVEL) de l'interprète consiste à évaluer indéfiniment la forme :

(WHILE T (TOPLEVEL))

C'est donc la fonction TOPLEVEL qui détermine le mode de fonctionnement de l'interprète. Cette fonction (de type SUBR) est bien évidemment redéfinissable par l'utilisateur qui désire se construire son propre système.

(TOPLEVEL) [SUBR à 0 argument]

Cette fonction va, par défaut :
- lire une S-expression dans le flux d'entrée courant
- évaluer cette S-expression
- imprimer le résultat de cette évaluation dans le flux de sortie courant

A l'initialisation du système la fonction TOPLEVEL est équivalente à :

```
(DE TOPLEVEL ()
    (TAG SYSERROR
         (PRINT '|= | (EVAL (READ)))))
```

ATTENTION : une redéfinition de cette fonction, qui n'évalue pas des formes lues, rend tout le système Le_Lisp 80 inutilisable.

exemple de définition à éviter :

(DE TOPLEVEL () (READ) (PRINT 'MARRE))

8.3 Le fichier initial

Au lancement du système Le_Lisp 80, un fichier initial est lu avant de passer en mode conversationnel sur le terminal. Ce fichier se nomme - LELISP.INI dans le système CP/M.

Ce fichier contiendra les définitions des EXPR, FEXPR ou MACRO que l'utilisateur désire posséder à chaque appel du système. En général ce fichier contient l'éditeur vidéo PEPE dans un format tassé (voir la fonction MAKE-INIT).

8.4 Le GARBAGE–COLLECTOR

La zone de l'interprète qui contient les objets Lisp est allouée dynamiquement. Quand cette zone est saturée une machinerie connue sous le nom de *garbage–collector* est automatiquement appelée pour récupérer les objets inutilisés.
Si cet essai s'avère infructueux, une erreur fatale se produit dont le libellé est :

```
** GC : zone liste pleine.
```

si la zone contenant les doublets de listes est pleine ou bien :

```
** GC : zone symbole pleine.
```

si c'est le cas de la zone allouée aux symboles.

(GC) [SUBR à 0 argument]

permet d'appeler le *garbage–collector* explicitement. GC retourne le nombre de doublets libres dans la zone liste.

(GCINFO) [SUBR à 0 argument]

permet de retourner une liste contenant les informations concernant la dernière récupération de mémoire. Cette liste a la forme :

```
(GC <n1> ATOM <n2> CONS <n3>)
```

<n1> est le nombre de GC depuis le début de la session, <n2> le nombre d'octets libres dans la zone symbole et <n3> le nombre de doublets libérés dans la zone liste.

(GCALARM <n>) [SUBR à 1 argument]

est une interruption programmée lancée automatiquement par le système après chaque récupération. L'argument <n> est le nombre de doublets libérés. Cette fonction permet d'éviter l'erreur fatale vue précédemment en provoquant une erreur normale pendant qu'il est encore temps.

Cette redéfinition de GCALARM permet, en cas de diminution trop importante du nombre de doublets de listes, de provoquer une erreur douce :

```
(DE GCALARM (n)
    (WHEN (< n 500)
          (PRINT '|J'arrete les frais|)
          (SYSERROR 'GCALARM '|nb de doublets restants| n)))
```

8.5 Les interruptions de l'interprète

Il est possible d'interrompre l'interprète durant une évaluation en tapant n'importe quel caractère sur le clavier. A ce moment le système appelle la fonction ITEVAL avec comme argument le code ASCII du caractère qui a provoqué l'interruption. Une redéfinition de cette fonction permet de résoudre le problème de la frappe anticipée.

(ITEVAL <n>) [SUBR à 1 argument]

est appelée par le système quand l'utilisateur frappe un caractère au clavier durant une évaluation.

Par défaut cette fonction est définie comme suit :

```
(DE ITEVAL (n)
    (SELECTQ n
        (#^ B (BREAK))
        (#^ C (END))
        (#^ S (TYI))
        (T ())))
```

Ce qui permet :
- de retourner à la boucle principale de l'interprète avec ^B,
- de suspendre une évaluation (en général durant une impression) avec ^S, tout appui sur une autre touche fera repartir cette impression
- de sortir définitivement du système Le_Lisp 80 avec ^C

Toutes les autres touches sont ignorées par défaut.

8.6 Les erreurs provoquées par l'interprète

A l'apparition d'une erreur durant une évaluation, un message décrivant l'erreur est édité dans le flux de sortie courant.
Ce message contient toujours le nom de la fonction qui provoque l'erreur, un message décrivant l'erreur elle-même et l'argument défectueux.
Pour réaliser ce travail, le système appelle toujours la fonction SYSERROR (qui est bien évidemment redéfinissable) qui elle-même doit appeler la fonction BREAK qui réalisera un échappement permettant de retourner à la boucle principale.

(SYSERROR <sym> <s1> <s2>) **[SUBR à 3 arguments]**

est appelée systématiquement par le système en cas d'erreur ou explicitement dans un programme. <sym> est le nom de la fonction qui a provoqué l'erreur. <s1> est le message décrivant l'erreur. <s2> est l'argument (ou la liste d'arguments) défectueux. Par défaut, SYSERROR imprime sur le flux de sortie courant le message :

```
** <sym> : <s1> : <s2>
```

Toutefois si le type de l'erreur <s1> est un symbole dont la valeur est définie, c'est cette valeur qui sera éditée dans le flux de sortie. Ceci permet de modifier rapidement les messages d'erreurs.

(BREAK) **[SUBR à 0 argument]**

indique l'action à faire après impression du message d'erreur. Cette fonction qui n'existe que pour être redéfinie permet de réaliser le mode mise-au-point (voir le chapitre suivant).

Voici la decription du traitement par défaut :

```
(DE SYSERROR (f m a)
    (PRINT '|** | f
           '| : | (IF (AND (SYMBOLP m)
                            (BOUNDP m))
                       (CVAL m)
                       m)
           '| : | a)
    (BREAK))

(DE BREAK () (EXIT SYSERROR 'SYSERROR))

; les libellés des erreurs standard

(SETQ
    ERSXT   '|erreur de syntaxe|
    ERIOS   '|erreur disque|
    EROOB   '|argument hors-limite|
```

```
ERUDV   '|variable indefinie|
ERUDF   '|fonction indefinie|
ERUDT   '|echappement indefini|
ERWNA   '|trop d'arguments|
ERWLA   '|liaison illegale|
ERNUM   '|CDR numerique|
ERNNA   '|l'argument n'est pas un nombre|
ERNVA   '|l'argument n'est pas une variable|
ERNAA   '|l'argument n'est pas un symbole|
ERNLA   '|l'argument n'est pas un CONS|

FERAT   '|zone des symboles pleine|
FERLS   '|zone des listes pleine|
FERFS   '|zone pile pleine|)
```

8.7 L'accès à la pile

(CSTACK <n>) [SUBR à 1 argument]

A chaque entrée dans une fonction, un échappement, un verrou ... l'interprète fabrique un bloc de contrôle dans la pile. CSTACK retourne les <n> derniers blocs de contrôle de la pile Lisp (ou tous les blocs si <n> = NIL). Chacun de ces blocs est transformé en une liste dont le CAR est le type du bloc, et le CDR les arguments du bloc. Voici la liste des blocs actuellement fabriqués par le système. Cette liste pourra s'allonger dans le futur au fur et à mesure du développement du système Le_Lisp 80.

type	argument
LAMBDA	F-val de la fonction
TAG	nom de l'échappement
FLET	corps
PROTECT	corps
LOCK	fonction
BIND	forme

Un exemple d'utilisation de cette fonction se trouve dans la fonction PRINTSTACK.

8.8 L'accès à la mémoire et au CPU

Certaines fonctions permettent d'accéder directement à la mémoire pour pouvoir définir de nouvelles fonctions SUBR, pour tester l'interprète ... Ces fonctions servent à construire le chargeur/assembleur du système Le_Lisp 80.

Ces fonctions utilisent des adresses mémoire quelconques. Les nombres entiers Le_Lisp 80 qui sont codés sur 14 bits ne permettent pas de représenter une adresse sur 16 bits. Une adresse mémoire <adr> sera donc représentée de 2 manières possibles :
- un nombre sur 14 bits pour les adresses comprises entre -2^13 et +2^13-1
- une liste de 2 nombres de la forme : (<high> <low>)

dans laquelle le premier nombre <high> contient l'octet de poids poids forts de l'adresse et le second <low> l'octet de poids faibles.

```
l'adresse #$8FFC   doit s'ecrire    (#$8F  #$FC)
```

(LOC <s>) **[SUBR à 1 argument]**

permet de connaître l'adresse absolue de l'objet Lisp <s>. L'adresse d'un objet Lisp est le pointeur sur la valeur de cet objet. Cette fonction permet de localiser en mémoire les objets Le_Lisp 80 qui sont crés dynamiquement.

```
ex : (LOC '(A B C))      ->  (40 26)
     (LOC '(A B C))      ->  (44 120)
```

(VAG <adr>) **[SUBR à 1 argument]**

est la fonction inverse de LOC. VAG retourne l'objet Lisp dont l'adresse absolue est fournie en argument. Donc (VAG (LOC <s>)) = <s>. Attention : VAG ne teste pas si <adr> est l'adresse d'un véritable objet Lisp, un appel erroné de cette fonction peut provoquer une erreur du système hôte.

(MEMORYB <adr> <n>) **[SUBR à 1 ou 2 arguments]**

permet de consulter ou de modifier (si le 2ème argument numérique <n> est fourni) n'importe quel octet de la mémoire dont l'adresse <adr> est fournie en premier argument. Cette fonction ne fait AUCUN contrôle de validité d'adresse et doit donc être utilisée avec beaucoup de précaution. MEMORYB retourne en valeur un nombre, sur 8 bits, représentant la valeur de l'octet après modification éventuelle.

(EXECUTE <l>) **[SUBR à 1 argument]**

<l> est une liste qui ne doit contenir que des nombres. Ces nombres représentent des instructions machine 8080 ou Z80. EXECUTE va charger cette liste en mémoire et lui passer le contrôle au moyen de l'instruction machine CALL. Cette liste d'instructions doit être terminée par l'instruction RET (#$C9) et retourner dans HL une valeur Lisp qui sera la valeur retournée par la fonction EXECUTE.

```
; Voici une fonction qui permet d'engendrer des
; nombres aléatoires dans l'intervalle 0 +127.
; (les commentaires sont de type Z80)

(DE RANDOM ()
    (EXECUTE [#$ED #$5F        ; LD A,R
              #$6F             ; LD L,A
              #$26 #$00        ; LD H,0
              #$C9]))          ; RET
```

(CALL <adr> <a1> <a2> <a3>) **[SUBR à 4 arguments]**

le premier argument <adr> doit être l'adresse d'un sous-programme en mémoire. CALL va lançer ce sous-programme après avoir charger les accumulateurs HL, DE et BC avec les valeurs respectives <a1>, <a2>, <a3>. Cette fonction doit être utilisée avec précaution car elle ne teste pas la validité du code qui est lançé.
CALL retourne en valeur la valeur actuelle de l'accumulateur HL après exécution du code débutant en <adr>.

Attention : le code invoqué au moyen de CALL doit impérativement se terminer par l'instruction machine, RET, et charger le registre HL avec une valeur Lisp qui est retournée par la fonction CALL.

(CALLN <adr> <l>) **[SUBR à 2 arguments]**

comme pour la fonction précédente, <adr> est une adresse en mémoire mais <l> représente une liste d'arguments évalués en nombre quelconque. CALLN permet d'appeler des SUBR à nombre d'arguments variables et doit être utilisée avec les mêmes précautions que la fonction CALL.

(IN <n>) **[SUBR à 1 argument]**

permet de lire la porte d'entrée du micro-processeur 8080 ou Z80 d'adresse <n>. la valeur qui y est lue est retournée sous la forme d'un nombre de 8 bits.

(OUT <n1> <n2>) **[SUBR à 2 arguments]**

permet d'envoyer sur la porte de sortie du micro-processeur 8080 ou Z80 de numéro <n1>, les 8 bits de poids faibles de la valeur <n2>. OUT retourne en valeur la valeur de <n2>.

$BCODE$ [Variable]

contient l'adresse du début d'une zone mémoire totalement libre qui peut être utilisée pour charger directement en binaire (au moyen de la fonction MEMORYB) de nouvelles fonctions.

$ECODE$ [Variable]

contient l'adresse de la fin de la zone précédente.

CHAPITRE 9

PISTAGE, MISE–AU–POINT ET PAS–A–PAS

Ce chapitre décrit plusieurs utilitaires, écrits en Lisp, qui permettent de mettre au point plus aiséement vos programmes Lisp. Ces fonctions sont chargées automatiquement à leur premier appel.

9.1 Le pistage

Il est possible tout d'abord de *pister* à l'exécution d'un programme tous les appels de la fonction interprète EVAL.

(TRACEVAL <s>) **[SUBR à 1 argument]**

est identique à la fonction EVAL toutefois à chaque appel interne de la fonction EVAL la forme à évaluer sera imprimée précédée du symbole --> et sa valeur du symbole <--.

Il est également possible de pister tous les appels d'une fonction de l'utilisateur. Ce pistage consiste à imprimer à l'entrée de la fonction, son nom ainsi que le nom et la valeur de chacun de ses arguments et à imprimer à sa sortie son nom et la valeur retournée par cette fonction.

(TRACE <sym1> ... <symN>) **[FEXPR]**

permet de pister les différentes fonctions <sym1> ... <symN>. Les noms de ces fonctions ne sont pas évalués. TRACE retourne la liste de toutes les fonctions traitées.

(UNTRACE <sym1> ... <symN>) **[FEXPR]**

permet d'enlever le pistage pour les différentes fonctions <sym1> ... <symN>. Les noms de ces fonctions ne sont pas évalués. Si aucun argument n'est donné (c'est-à-dire si on invoque (UNTRACE), enlève le pistage de toutes les fonctions. UNTRACE retourne la liste de toutes les fonctions traitées.

(TRACEND) **[EXPR à 0 argument]**

récupère la place occupée par les fonctions de pistage. Retourne le symbole TRACEND en valeur.

```
; exemple d'utilisation du pisteur

; définition d'une fonction

? (de rv (l r)
?     (if (null l)
?         r
?         (rv (cdr l) (cons (car l) r))))
= RV
?
; essai normal sans pistage
? (rv '(x y z))
= (Z Y X)
?
; appel du pisteur sur RV
? (trace rv)
Je charge : TRACE   LL .....
= (RV)
?
; essai de la fontion pistée
? (rv '(a b c d))
RV ---> L = (A B C D)  R = NIL
RV ---> L = (B C D)  R = (A)
RV ---> L = (C D)  R = (B A)
RV ---> L = (D)  R = (C B A)
RV ---> L = NIL  R = (D C B A)
RV <--- (D C B A)
RV <--- (D C B A)
RV <--- (D C B A)
RV <--- (D C B A)
RV <--- (D C B A)
= (D C B A)
?
; enlève le pistage
? (untrace)
= (rv)
?
```

9.2 Le mode mise au point

Quand une erreur se déclenche un message est imprimé puis l'environnement est restauré jusqu'au premier (TAG SYSERROR ... rencontré ce qui peut amener la réinitialisation de tous les objets dynamiques (variables, fonctions ...). Or il est parfois souhaitable, quand une erreur se produit, de rester dans l'environnement de l'erreur pour contrôler les variables ou la pile *au moment de l'erreur*. Ceci est réalisé en passant en mode mise-au-point.

(DEBUG <s>) [FEXPR]

Si l'argument <s>, qui n'est pas évalué, est égal à T, passage en mode mise-au-point globalement. Si l'argument vaut NIL, le mode mise-au-point est désactivée globalement. Enfin si l'argument <s> est une expression quelconque, le mode mise-au-point ne sera actif que durant l'évaluation de cette expression. DEBUG retourne en valeur la valeur de l'évaluation de <s> (ce qui permet d'encapsuler temporairement toute expression Lisp avec la fonction DEBUG pour la mettre au point).

Quand ce mode est activé par une erreur, le message suivant apparait :

```
vous etes dans la fonction BREAK.  ^D pour sortir.
```

puis les 8 derniers niveaux de la pile sont imprimés. Il est possible à de moment d'inspecter voire de modifier le contenu des variables, des fonctions, de visualiser la pile en entier ... Le retour à la boucle principale s'effectue par l'évaluation de la forme (EXIT BREAK) ou en abrégé ^D. Si une nouvelle erreur se produit dans la boucle d'inspection, BREAK est appelée récursivement. Cela est indiqué par le caractère d'invite qui contient autant de > que de niveaux d'appel de BREAK.

```
?
? (de cnt (s)
?     (if (atom s)
?         x
?         (+ (cnt (car s)) (cnt (cdr s)) 1)))
= cnt

? (cnt '(a b))
** EVAL : variable indefinie : X
   4    BREAK
   3    CNT
   2    CNT
   1    (LOCK UNDEF)
vous etes dans la fonction BREAK.  ^D pour sortir.

>? s
= A
```

```
>? x

** EVAL : variable indefinie : X

   8      BREAK
   7      (TAG SYSERROR)
   6      (TAG BREAK)
   5      (BIND (((PROMPT (CONCAT '> (PROMPT)))) (PRINT
   4      BREAK
   3      CNT
   2      CNT
   1      (LOCK UNDEF)
vous etes dans la fonction BREAK.  ^D pour sortir.

> > ?  ^D

>?  ^D

?
```

(PRINTSTACK <n>) **[EXPR à 1 argument]**

visualise les <n> derniers niveaux de la pile de l'interprète ou de toute cette pile si l'argument n'est pas fourni.

(DEBUGEND) **[EXPR à 0 argument]**

permet de récupérer la place occupée par les fonctions du mode mise-au-point.

9.3 L'exécution pas à pas

L'exécution en mode pas à pas permet de s'arrêter à chaque appel interne de l'évaluateur, ainsi qu'à chacun de ses retours. Durant ces arrêts l'utilisateur peut consulter la valeur des objets dynamiques et suivre très finement le déroulement de ses évaluations.

Pour réaliser cette exécution incrémentale le système, à chaque appel interne de l'évaluateur, invoque la fonction STEPIN avec comme argument la forme à évaluer et au retour de l'évaluation la fonction STEPOUT avec la valeur de cette forme. Attention, la fonction STEPIN doit retourner une nouvelle forme qui sera évaluée sans arrêt.

Voici les différentes actions à chaque arrêt :

> continuer d'évaluer l'expression courante en mode pas à pas.

< évaluer l'expression courante sans arrêt et reprendre le mode pas à pas au retour de cette évaluation.

. arrêter le mode pas à pas et retourner la valeur de l'expression de départ.

= appeler une fois la fonction TOPLEVEL, ce qui permet d'évaluer n'importe quelle expression.

(STEP <s>) **[EXPR à 1 argument]**

évalue l'expression <s> en mode pas à pas. Retourne la valeur de l'évaluation de <s>.

(STEPEND) **[EXPR à 0 argument]**

récupère l'espace occupé par les fonctions de l'exécution pas à pas.

CHAPITRE 10

LES FONCTIONS UTILES

Ce chapitre contient les définitions d'un certain nombre de fonctions qui n'existent pas dans le système Le_Lisp 80 pour des raisons d'encombrement mémoire, mais dans d'autres systèmes Le_Lisp.

(COMMENT <e1> ... <eN>) **[FEXPR]**

retourne le symbole COMMENT lui-même sans évaluer aucun des arguments. Cette fonction est très utile pour rendre ineffective une S-expression quelconque à l'intérieur d'un fichier mais est à déconseiller pour l'introduction de véritables commentaires qui sont plutot écrits précédés du caractère spécial point-virgule.
ATTENTION : cette fonction ne peut être placé qu'à l'intérieur d'un PROGN (implicite ou explicite) sous peine de perturber l'évaluation.

COMMENT peut être défini en Lisp de la manière suivante :

```
(DF COMMENT 1 'COMMENT)

ex : (COMMENT 'FOO BAR)    -> COMMENT
     (COMMENT)             -> COMMENT
     (COMMENT MAIS NON)    -> COMMENT
```

(IDENTITY <s>) **[EXPR à 1 argument]**

comme son nom l'indique, cette fonction est la fonction identité et retourne comme il se doit son argument.

IDENTITY peut être défini en Lisp de la manière suivante :

```
(DE IDENTITY (s) s)

ex : (IDENTITY 'A)        -> A
     (IDENTITY (1+ 5))    -> 6
```

(MAKELIST <n> <s>) **[EXPR à 2 arguments]**

retourne une liste de <n> éléments. Chacun de ces éléments contient la valeur <s>. Bien utile pour construire des listes de longueur donnée.

MAKELIST peut être défini en Lisp de la manière suivante :

```
(DE MAKELIST (n s)
    (IF (<= n 0)
        ()
        (CONS s (MAKELIST (1- n) s))))

ex : (MAKELIST 4 'A)   ->  (A A A A)
```

(CATCH-ALL-BUT <l> <s1> ... <sN>) [FEXPR]

<l> est une liste de noms d'échappements. Si durant l'évaluation des expressions <s1> ... <sN> un appel d'échappement est réalisé (au moyen d'un appel de EXIT ou EVEXIT) qui provoque une sortie du CATCH-ALL-BUT alors cette fonction teste si le nom de l'échappement en question fait partie de la liste de noms d'échappement <l>. S'il en fait partie l'échappement est réalisé normalement, s'il n'en fait pas partie, l'erreur *échappement indéfini* est déclenchée. Cette fonction permet donc de *filtrer* les échappements.

CATCH-ALL-BUT peut être défini en Lisp de la manière suivante :

```
(DF CATCH-ALL-BUT (ltag . l)
    (LOCK (LAMBDA (tag val)
            (IFN tag
                 val
                 (IFN (MEMQ tag ltag)
                      (SYSERROR 'CATCH-ALL-BUT 'ERUDT tag)
                      (EVEXIT tag val))))
          (EPROGN l)))
```

(FALSE) [EXPR à 0 argument]

retourne la valeur booléenne fausse, c'est-à-dire la constante NIL.

(TRUE) [EXPR à 0 argument]

retourne la valeur booléenne vraie, c'est-à-dire la constante T.

(XCONS <s1> <s2>) [EXPR à 2 arguments]

cette fonction est équivalente à :

```
(CONS <s2> <s1>)
```

(NCONS <s>) [EXPR à 1 argument]

cette fonction correspond à :

```
(CONS <s> NIL)
```

(FIRSTN <n> <l>) **[EXPR à 2 arguments]**

retourne une copie des <n> premiers éléments de le liste <l>. Si la liste <l> possède moins de <n> éléments (c'est-à-dire si <n> > (LENGTH <l>)), FIRSTN retourne une copie de toute la liste <l>.

FIRSTN peut être défini en Lisp de la manière suivante :

```
(DE FIRSTN (n l)
    (COND ((NULL l) ())
          ((<= n 0) ())
          (t (CONS (CAR l) (FIRSTN (1- n) (CDR l))))))

ex : (FIRSTN 3 '(A B C D E F))    ->  (A B C)
     (FIRSTN 5 '(A B C D))        ->  (A B C D)
```

(LASTN <n> <l>) **[EXPR à 2 arguments]**

retourne une copie des <n> derniers éléments de la liste <l>. Si la liste contient moins d'éléments que demandé (c'est-à-dire si (LENGTH l) < n) LASTN retourne une copie de la liste <l> entière.

LASTN peut être défini en Lisp de la manière suivante :

```
(DE LASTN (n l)
    (REVERSE (FIRSTN n (REVERSE l))))

ex : (LASTN 2 '(A B C D E))   ->  (D E)
     (LASTN 10 '(A B C))      ->  (A B C)
```

(REMQ <sym> <l>) **[EXPR à 2 arguments]**

retourne une copie de <l> dans laquelle toutes les occurrences du symbole <sym> ont été enlevées.

REMQ peut être défini en Lisp de la manière suivante :

```
(DE REMQ (sym l)
    (COND ((NULL l) ())
          ((EQ sym (CAR l)) (REMQ sym (CDR l)))
          (T (CONS (CAR l) (REMQ sym (CDR l))))))

ex : (REMQ 'A '(A B A (C A B) D A S))   ->  (B (C A B) D S)
```

(REMOVE <s> <l>) **[EXPR à 2 arguments]**

cette fonction est identique à la fonction précédente mais réalise le test au moyen du prédicat EQUAL.

REMOVE peut être défini en Lisp de la manière suivante :

```
(DE REMOVE (s l)
    (COND ((NULL l) ())
          ((EQUAL s (CAR l)) (REMOVE s (CDR l)))
          (T (CONS (CAR l) (REMOVE s (CDR l))))))

ex : (REMOVE '(A) '(A B (A) (C A B) (A) S))
             ->  (A B (C A B) S)
```

(NREVERSE <l>) **[EXPR à 1 argument]**

renverse physiquement et rapidement la liste <l>. Cette fonction doit être manipulée avec précaution car elle modifie physiquement toute la liste ce qui peut provoquer des catastrophes si cette liste était partagée.

NREVERSE peut être défini en Lisp de la manière suivante :

```
(DE NREVERSE (l)
    (IF (CONSP l)
        (LET ((l l) (r ()))
             (IF (NULL (CDR l))
                 (RPLACD l r)
                 (SELF (CDR l) (RPLACD l r))))
        l))
```

(NSUBST <s1> <s2> <l>) **[EXPR à 3 arguments]**

modifie physiquement la liste <l> en substituant à chaque occurrence de l'expression <s2>, l'expression <s1>. Cette fonction utilise le prédicat EQUAL pour réaliser le test. Pour obtenir une copie de la liste substituée il faut utiliser la fonction SUBST.

NSUBST peut être défini en Lisp de la manière suivante :

```
(DE NSUBST (new old l)
    (COND ((EQUAL l old) new)
          ((ATOM l) l)
          (T (RPLAC l
                    (NSUBST new old (CAR l))
                    (NSUBST new old (CDR l))))))

ex : (SETQ l '(A C (D A)))    ->  (A C (D A))
     (NSUBST '(X Y Z) 'A l)   ->  ((X Y Z) C (D (X Y Z)))
     l                        ->  ((X Y Z) C (D (X Y Z)))
```

(DELQ <sym> <l>) **[EXPR à 2 arguments]**

enlève physiquement toutes les occurrences du symbole <sym> au premier niveau de la liste <l>. Cette fonction utilise le prédicat EQ. Pour réaliser une copie de ce type de liste, il faut utiliser la fonction REMQ.

DELQ peut être défini en Lisp de la manière suivante :

```
(DE DELQ (sym l)
    (COND ((ATOM l) l)
          ((EQ sym (CAR l)) (DELQ sym (CDR l)))
          (T (RPLACD l (DELQ sym (CDR l))))))

ex : (SETQ l '(A B C B B D))  ->  (A B C B B D)
     (DELQ 'B l)              ->  (A C D)
```

(DELETE <s> <l>) [EXPR à 2 arguments]

enlève physiquement toutes les occurrences de l'expression <s> au premier niveau de la liste <l>. Cette fonction utilise le prédicat EQUAL. Pour fabriquer une copie de ce type de liste, il faut utiliser la fonction REMOVE.

DELETE peut être défini en Lisp de la manière suivante :

```
(DE DELETE (s l)
    (COND ((ATOM l) l)
          ((EQUAL s (CAR l)) (DELETE s (CDR l)))
          (T (RPLACD l (DELETE s (CDR l))))))

ex : (SETQ l '(A (B) C B (B) D))  ->  (A (B) C B (B) D)
     (DELETE '(B) l)              ->  (A C B D)F
```

(RASSQ <sym> <al>) [EXPR à 2 arguments]

retourne l'élément de la A-liste <al> dont la valeur (le CDR) est égal au symbole <sym>, sinon RASSQ retourne NIL.

RASSQ peut être défini en Lisp de la manière suivante :

```
(DE RASSQ (sym al)
    (COND ((NULL al) NIL)
          ((EQ (CDAR al) at) (CAR al))
          (T (RASSQ sym (CDR al)))))
```

(ASSOC <s> <al>) [EXPR à 2 arguments]

Cette fonction est identique à la fonction ASSQ mais utilise le prédicat EQUAL pour tester les clefs qui peuvent donc être de n'importe quel type.

ASSOC peut être défini en Lisp de la manière suivante :

```
(DE ASSOC (s al)
   (COND
      ((NULL al) NIL)
      ((EQUAL s (CAAR al)) (CAR al))
      (T (ASSOC s (CDR al)))))
```

(CASSOC <s> <al>) **[EXPR à 2 arguments]**

Cette fonction est identique à la fonction CASSQ mais utilise le prédicat EQUAL pour tester les clefs qui peuvent donc être de n'importe quel type.

(CASSOC s l) est donc équivalent à (CDR (ASSOC s l))

(RASSOC <s> <al>) **[EXPR à 2 arguments]**

cette fonction est équivalente à la fonction RASSQ mais utilise le prédicat EQUAL pour tester les valeurs.

RASSOC peut être défini en Lisp de la manière suivante :

```
(DE RASSOC (s al)
    (COND ((NULL al) NIL)
          ((EQUAL s (CDAR al)) (CAR al))
          (T (RASSOC a (CDR al)))))
```

(PAIRLIS <l1> <l2> <al>) **[EXPR à 3 arguments]**

<l1> doit être une liste de clefs
<l2> doit être une liste de valeurs
PAIRLIS retourne une nouvelle A-liste formée à partir de la liste des clefs <l1> et de la liste des valeurs associées <l2>. Si le troisième argument <al> est fourni, il est ajouté à la fin de la A-liste créée.

PAIRLIS peut être défini en Lisp de la manière suivante :

```
(DE PAIRLIS (l1 l2 al)
   (IF (NULL l1)
       al
       (CONS (CONS (CAR l1) (CAR l2))
          (PAIRLIS (CDR l1) (CDR l2) al))))

ex : (PAIRLIS '(YA JESTEM WIELKY)  '(LE MERLE CHANTE) ())
       -> ((YA . LE) (JESTEM . MERLE) (WIELKY . CHANTE))
     (PAIRLIS '(X Y Z) '(A (B)) '((A . X) (B . Y)))
       -> ((X . A) (Y B) (Z) (A . X) (B . Y))
```

(PSETQ <sym1> <s1> ... <symN> <sN>) **[FEXPR]**

identique à la fonction SETQ mais les affectations ont lieu *en parallèle*. Très utile pour réaliser des permutations de variables. PSETQ retourne <s1> en valeur.

PSETQ peut être défini en Lisp de la manière suivante :

```
(DF PSETQ l
    (LET ((lvar) (lval))
         (WHILE l (NEWL lvar (NEXTL l))
```

```
                        (NEWL lval (EVAL (NEXTL l))))
                (MAPC 'SET lvar lval)))

; ou sous la forme de la macro

    (DM PSETQ l
        (WHEN (CDR l)
              ['SETQ (CADR l)
                     ['PROG1 (CADDR l)
                             ['PSETQ (CDDDR l)]]])))

ex : (SETQ x 10 y 11 z 12)  ->  12
     (PSETQ x y y z z x)    ->  11
     x                      ->  11
     y                      ->  12
     z                      ->  10
```

(DESET <l1> <l2>) [EXPR à 2 arguments]

réalise le *destructuring SET* du système *NIL* [White 79]. <l1> est un arbre de variables, <l2> est un arbre de valeurs. DESET va affecter aux variables de l'arbre des variables les valeurs correspondantes de l'arbre de valeurs. DESET retourne toujours T en valeur.

DESET peut être défini en Lisp de la manière suivante :

```
    (DE DESET (l1 l2)
        (COND ((NULL l1)
               (OR (NULL l2)
                   (SYSERROR 'DESET '|trop de valeurs| l2)))
              ((AND l1 (SYMBOLP l1)) (SET l1 l2) T)
              ((AND (CONSP l1) (LISTP l2))
               (DESET (CAR l1) (CAR l2))
               (DESET (CDR l1) (CDR l2)))
              (T (SYSERROR 'DESET '|liaison impossible|
                           (LIST l1 l2)))))

ex : (DESET '(A (B . C)) '((1 2) (3 4)))    ->  T
     A                                      ->  (1 2)
     B                                      ->  3
     C                                      ->  (4)
```

(DESETQ <l1> <l2>) [FEXPR]

est équivalent à la fonction précédente mais le 1er argument n'est pas évalué (à la SETQ).

```
(DESETQ <l1> <l2>)
                    ; est donc equivalent a
(DESET (QUOTE <l1>) <l2>)
```

(READLINE) **[EXPR à 0 argument]**

lit la ligne suivante du flux d'entrée et retourne la liste des caractères de cette ligne. Cette liste ne contient pas le caractère fin de ligne (en l'occurrence le caractère Return) ni les caractères line-feed. READLINE est utilisé pour réaliser à peu de frais les interfaces entre programme et utilisateur.

ATTENTION : cette fonction n'effectue pas de conversion automatique des caractères minuscules en caractères majuscules.

READLINE peut être défini en Lisp de la manière suivante :

```
(DE READLINE ()
    (LET ((l) (c))
         (WHILE (NEQ (SETQ c (READCN)) #^M)
                (NEWL l c))
         (NREVERSE l)))
```

```
ex :  ? (MAPCAR 'ASCII (READLINE))
                                  ; appelle la lecture d'une ligne
      Le gai rossignol            ; ligne terminee par Return
      = (L e !  g a i !  r o s s i g n o l )
                                  ; liste des caracteres lus

      ; pour fabriquer une suite de mots a partir d'une ligne
      ; qui ne contient pas de caracteres speciaux ( ) .

      ? (IMPLODE (APPEND #/( (READLINE) #/))))
      Le merle moqueur                ; ligne terminee par Return
      = (LE MERLE MOQUEUR)            ; liste des atomes lus
```

(GETFN <sym>) **[EXPR à 1 argument]**

permet de retourner la lambda-expression (de type LAMBDA, FLAMBDA ou MLAMBDA) associée au symbole <sym>.

```
ex :  (DE BAR (n) (+ n n))      ->  BAR
      (TYPEFN 'BAR)             ->  EXPR
      (GETFN 'BAR)              ->  (LAMBDA (n) (+ n n))
      (DM FOO (x) (CAR x))      ->  FOO
      (TYPEFN 'FOO)             ->  MACRO
      (GETFN 'FOO)              ->  (MLAMBDA (x) (CAR x))
```

(NOT <s>) **[EXPR à 1 argument]**

effectue l'inversion de la valeur booléenne <s>. Du fait de la construction des valeurs booléennes en Lisp, cette fonction est identique à la fonction NULL.

(TAILP <s> <l>) [EXPR à 2 arguments]

si <s> est un des CDR de <l>, TAILP retourne <s> sinon T.

(TAILP <s> <l>) [EXPR à 2 arguments] peut être défini en Lisp de la manière suivante :

```
(DE TAILP (s l)
    (COND ((ATOM l) ())
          ((EQ s l) s)
          (T (TAILP s (CDR l)))))
```

```
ex : (SETQ l '(A B C D) ll (CDDR l))   ->   (C D)
     (TAILP ll l)                      ->   (C D)
     (TAILP '(C D) '(A B C))           ->   NIL
     ; car TAILP utilise EQ
```

CHAPITRE 11

L'ENVIRONNEMENT STANDARD

11.1 Le fichier lanceur

```
;================================================================
;
;              Le Lisp 80 v12  :  Le Fichier d'Initialisation
;
;================================================================
;  (c) 1982    Jérôme CHAILLOUX   et   I.N.R.I.A.
;              B.P. 105, 78153 Le Chesnay Cédex France
;================================================================


;=====  Le chargement initial (et silencieux)

(progn

  ;-----  Test du numéro de version

  #.(unless (= (version) 12)
            (syserror 'startup 'version (version)))

  ;-----  Réalisation du chargement

  (untilexit eof
       (tag syserror (eval (read)))
       (tyo #/.))
  (input)
  (terpri)
  (setq read () toplevel () eval ())
  (tyo '#"Systeme : " (explode (system))
       '#"  Version : " (explode (version))
       '#"  Memoire : ")
  (tyo (explode (gc))
       #^ M #^ J)
  T))))
```

```
;=====   Les noms des erreurs

(SETQ

  ERSXT   '|erreur de syntaxe|
  ERIOS   '|erreur disque|
  EROOB   '|argument hors limite|
  ERUDV   '|variable indefinie|
  ERUDF   '|fonction indefinie|
  ERUDT   '|echappement indefini|
  ERWNA   '|trop d'arguments|
  ERWLA   '|liaison illegale|
  ERARI   '|erreur arithmetique|
  ERNUM   '|CDR numerique|
  ERNVA   '|l'argument n'est pas une variable|
  ERNAA   '|l'argument n'est pas un symbole|
  ERNNA   '|l'argument n'est pas un nombre|
  ERNLA   '|l'argument n'est pas une liste|

  FERAT   '|zone des symboles pleine|
  FERLS   '|zone des listes pleine|
  FERFS   '|zone pile pleine|
)

;-----   les MACROs LET et LETN

(dm let l
    (rplac l
           (mcons lambda (mapcar 'car (cadr l)) (cddr l))
           (mapcar 'cadr (cadr l))))

(dm letn (letn sym larg . corps)
    ['flet [[sym (mapcar 'car larg)  corps]]
           [sym  (mapcar 'cadr larg)]])

;----   LET paragraphé comme WHILE

(ptype 'let 2)
(ptype 'letn 2)

;-----   Vérificateur de nom de fichier

(de  filename (f ext)
     ; rajouter l'extension .LL par defaut
     (if (> (plength f) 8)
         f
         (setq f (explode f))
         (repeat (- 8 (length f)) (newr f #/ ))
         (nconc f (explode (or ext 'LL)))
         (repeat (- 11 (length f)) (newr f #/ ))
         (implode (append '#"|" f '#"|"))))
```

```
(de selectdrive (d)
    ; change le disque courant de CP/M
    (execute [#$1E (- (cascii d) #/A)     ; MVI  E, 0/1/2 ...
             #$0E 14                       ; MVI  C,code
             #$CD 05 00])                  ; CALL 0005
     d)

;-----  LOAD et AUTOLOAD

(df load (f)
    (loadfile f)))

(de  loadfile (l)
     ; chargement en douceur d'un fichier disque
     (setq l (filename l))
     (ifn (input l)
          (list l '| n'existe pas|)
          (tyo '#"Je charge : " (explode l))
          (untilexit eof
                  (eval (read))
                  (tyo #/.))
          (input)
          (terpri)
          l)))

(dmc |%| () ['load (read)]))

(df AUTOLOAD (f . l)
      ; definition de fonction autoload
      ; (AUTOLOAD fichier syml ... symN)
      (MAPC (LAMBDA (A)
               (EVAL ['DM A 'L
                         ['REMFN (KWOTE A)]
                         ['LOAD f]
                         'L]))
            l))

(progn
      (AUTOLOAD PRETTY PRETTY PRETTYEND)
      (AUTOLOAD IMPR IMPR IMPRTOUT)
      (AUTOLOAD TRACE TRACE UNTRACE)
      (AUTOLOAD DEBUG DEBUG STEP)
      (AUTOLOAD SORTL SORTL)
      (AUTOLOAD VDT VDT VDTEND)
      (AUTOLOAD HANOI HANOI HANOIEND)
      (AUTOLOAD PEPE PEPE PEPEND)
)

;-----  Le macro-caractère d'appel de PEPE

(dmc |&| () ['PEPE (read)])
```

```
;=====   Fabrication du fichier  LELISP.INI

(de  make-init (i)
     ; fabrique le fichier initial en format tasse,
     ; sans les commentaires ni les renfoncements.
     (output '|LELISP  INI|)
     (bind ((sprint t))
           (mapc (lambda (f)
                    (input f)
                    (untilexit eof (print (read)))
                    (input))
                 (ifn i
                      '(|STARTUP LL | |VIRTTY  LL |)
                      '(|STARTUP LL | |VIRTTY  LL | |PEPE    LL |
                        |PEPTOP  LL | |PEPLN2  LL | |PEPMSC  LL |
                        |PEPCMD  LL | |PEPESC  LL | |PEPINI  LL |)))
     (output))))
```

11.2 La description du terminal virtuel

```
;=============================================================
;
;   Le Lisp 80 v12  :   VIRTTY  -  gestion du terminal virtuel
;
;=============================================================
;  (c) 1982    Jérôme CHAILLOUX    et    I.N.R.I.A.
;              B.P. 105, 78153 Le Chesnay Cédex France
;=============================================================

;  Ce fichier contient toutes les fonctions de contrôle de l'écran
;  dépendantes du type du terminal (ici du système Le Lisp 80).
;  Pour chaque nouveau système, il faut définir ses nouvelles
;  caractéristiques en mettant à jour ce fichier.


;-----   La taille physique de l'écran (compté à partir de 0)

#.(selectq (system)

    (MICRAL      '(setq $xmax$ 79  $ymax$ 23))
    (SILZ        '(setq $xmax$ 79  $ymax$ 23))
    (LX529E      '(setq $xmax$ 79  $ymax$ 23))
    (t           '(setq $xmax$ 79  $ymax$ 23))))


;-----   Le code ASCII des touches flèches
```

```
#.(selectq (system)

  (MICRAL  '(setq $left$ #^ H $right$ #^ I
                  $up$ #^ K $down$ #^ J))
  (SILZ    '(setq $left$ #^ H $right$ #^ L
                  $up$ #^ K $down$ #^ J))
  (LX529E  '(setq $left$ #^ H $right$ #^ Q
                  $up$ #^ S $down$ #^ R))
  (t       '(setq $left$ #^ B $right$ #^ F
                  $up$ #^ P $down$ #^ N)))


;-----  Les fonctions physiques sur l'écran :

#.(selectq (system)


   (MICRAL '(progn

          (de tycursor (x y)
              (tyo #$1B #/F (+ #$20 x) (+ #$20 y)))
          (de tybeep () (tyo #^ G))
          (de tycls () (tyo #^ L))
          (de tycleol () (tyo #$1B #/K))
          (de tycleos () (tyo #$1B #/J))
          (de tyinsch (n) ())
          (de tydelch () ())
          (de tyinsln () (tyo #$1B #/L))
          (de tydelln () (tyo #$1B #/M))
          (de tyattrib (i) (tyo #$1B (if i #/3 #/4)))

     ))

   (SILZ '(progn

        (de tycursor (x y)
            (tyo #$1B #/= (+ #$20 y) (+ #$20 x)))
        (de tybeep () (tyo #^ G))
        (de tycls () (tyo #^ Z))
        (de tycleol () (tyo #^ X))
        (de tycleos () (tyo #^ Q))
        (de tyinsch (n) ())
        (de tydelch () ())
        (de tyinsln () ())
        (de tydelln () ())
        (de tyattrib (i) (execute [#$3E (if i 142 143)
                                   #$C3 #$0F #$F0]))

     ))

   (LX529E '(progn
```

```
        (de tycursor (x y) (tyo 9 x 11 y))
        (de tybeep () (tyo #^ G))
        (de tycls () (tyo #^ Z))
        (de tycleol () (tyo 1 22))
        (de tycleos () ())
        (de tyinsch (n) ())
        (de tydelch () ())
        (de tyinsln () ())
        (de tydelln () ())
        (de tyattrib (i) (tyo (if i 15 14)))

    ))

  (t (syserror 'VIRTTY (system) (version) ))))))))))

;-----  Les fonctions communes à tous les systèmes

(de tyco (x y . l)
    ; positionne le curseur et fait un TYO
    (tycursor x y)
    (apply 'tyo l))
```

11.3 Programmes de démonstration

```
;=============================================================
;
;            Le Lisp 80 v12  :  Les Tours de Hanoi Animées
;
;=============================================================
;  (c) 1982    Jérôme CHAILLOUX   et   I.N.R.I.A.
;            B.P. 105, 78153 Le Chesnay Cédex France
;=============================================================


(de init-disque ()
    (setq l-disque
      '(#"          |          "
        #"         *|*         "
        #"        **|**        "
        #"       ***|***       "
        #"      ****|****      "
        #"     *****|*****     "
        #"    ******|******    "
        #"   *******|*******   "
        #"  ********|********  "
        #" *********|********* "
```

```
        #"====================="
        #"                     ")))
      ()))
```

;----- La fonction principale

```
(de hanoi (n)
    (unless n
            (print '|Combien de disque |)
            (setq n (read)))
    (unless (and (numberp n) (> n 2) (< n 10))
            (syserror 'hanoi 'eroob n))
    (let ((a1 '(10))        ; contenu de la 1ere aiguille
          (a2 '(10))        ; contenu de la 2eme aiguille
          (a3 '(10))        ; contenu de la 3eme aiguille
          (nmv 0)           ; nombre de mouvements
          (l-disque))
         (tycls)
         (putprop 'a1 0 'pos)
         (putprop 'a2 25 'pos)
         (putprop 'a3 50 'pos)
         (init-disque)
         (let ((n n)) (repeat n (newl a1 n) (decr n)))
         (tyattrib t)
         (tyco 28 1 '#"Les Tours de Hanoi")
         (tyattrib ())
         (affiche 'a1)
         (affiche 'a2)
         (affiche 'a3)
         (hanoi-moteur n 'a1 'a2 'a3)
         (tycursor 0 #.(1- $ymax$))
         'HANOI)))

(de hanoi-moteur (n dep arr int)
    (when (> n 0)
          (hanoi-moteur (1- n) dep int arr)
          (bouge n dep arr)
          (hanoi-moteur (1- n) int arr dep))))

(de bouge (n dep arr)
    (tycursor 18 5)
    (tyo (explode (incr nmv))
         '#" : je bouge le disque " (explode n)
         '#" de " (explode dep) '#" vers " (explode arr))
    (monte n (getprop dep 'pos) (- 10 (length (eval dep))))
    (avance n (getprop dep 'pos) (getprop arr 'pos))
    (descend n (getprop arr 'pos) (- 10 (length (eval arr))))
    (set dep (cdr (eval dep)) arr (cons n (eval arr))))))
```

;----- La fonction d'affichage

```
(de affiche (a)
    ; affiche toute l'aiguille "a"
    (let ((y 21)
          (l (reverse (eval a)))
          (x (getprop a 'pos)))
         (repeat 10
                 (tyco x (decr y)
                       (nth (or (nextl l) 0) l-disque)))
         (tyco (+ x 10) 21 (explode a))))

;-----  Les fonctions de mouvement de disque

(de monte (n x nb)
    ; disque "n" en "x" "nb" fois
    (let ((y (+ nb 10)))
         (repeat (1+ nb)
                 (tyco x y (nth n l-disque))
                 (tyco x (1+ y) (car l-disque))
                 (decr y)))))))))))

(de descend (n x nb)
    ; disque "n" en "x" "nb" fois
    (let ((y 11))
         (tyco x 10 (nth 11 l-disque))
         (repeat (1- nb)
                 (tyco x y (car l-disque))
                 (tyco x (incr y) (nth n l-disque))))))

(de avance (n x1 x2)
    ; avance horizontalement le disque "n"
    ; de la colonne "x1" a la colonne "x2"
    (if (> x1 x2)
        (repeat (- x1 x2)
                (tyco (decr x1) 10 (nth n l-disque))))
        (repeat (- x2 x1)
                (tyco (incr x1) 10 (nth n l-disque)))))))


;-----  Pour récuperer la place occupée par HANOI

(de hanoiend ()
    (mapc 'remfn
          '(init-disque hanoi hanoi-moteur
            bouge affiche monte descend avance))
    (autoload hanoi hanoi)
    'hanoiend)


;=============================================================
;
;      Le Lisp 80 v12  :   (V)er (D)e (T)erre   de type LOMBRIC
;
```

```
;=============================================================
;  (c) 1982    Jérôme CHAILLOUX    et    I.N.R.I.A.
;                B.P. 105, 78153 Le Chesnay Cédex France
;=============================================================


;-----  Petite démonstration de l'utilisation des fonctions
;        du terminal virtuel :  VIRTTY.LL  et de l'art de
;        l'animation video et de sa problematique en Lisp :
;                - éviter les garbage-collections
;                - vitesse indépendante de la taille des objets


(de vdt ()
    (let ((dir 'd)        ; la direction du ver (en symbolique).
          (lobst)         ; la liste des obstacles
          (chemin)        ; le chemin du ver.
          (ochemin)       ; le chemin du coup precedent.
          (taille)        ; la taille du ver.
          (x #.(// $xmax$ 2))  ; la position actuelle de la
          (y #.(// $ymax$ 2))  ;     tete du lombric.
          (xy)            ; la position packee.
          (score 0)       ; comme son nom l'indique.
          (mscore 0)      ; les milliers (du score)
          (commande))     ; une commande qui n'a pas ete traitee
          ;
          ; 1) dessine le cadre
          ;
          (tycls)
          (tyattrib ())
          (tyco 10 0 '#"***   V . D . T .   ***     Score :")
          (tycursor 0 1)
          (repeat $xmax$ (tyo #/.))
          (tycursor 0 $ymax$)
          (repeat $xmax$ (tyo #/.))
          (let ((y 1))
               (repeat (1- $ymax$)
                  (tyco 0 y #/.)
                  (tyco $xmax$ y #/.)
                  (incr y)))
          ;
          ; 2) fabrique le ler lombric et la liste des obstacles.
          ;
          (repeat (setq taille 5)
                  (newl chemin (+ (* y 128) (incr x))))
          (mapc (lambda (x y)
                        (newl lobst (+ (* y 128) x))
                        (tyco x y #/*))
                [10 10 #.(- $xmax$ 10) #.(- $xmax$ 10) #.(// $xmax$ 2)]
                [5 #.(- $ymax$ 5) 5 #.(- $ymax$ 5) #.(// $ymax$ 2)]
          )
```

```
;
; 3) la boucle principale
;
(flet ((iteval (n) (setq commande n)))
  ; une petite attente ... pour se preparer
  (repeat 5 (gc))
  (untilexit vdt
    ; decodage de la commande
    (selectq (if (null commande)
                 (tys)
                 (progl commande (setq commande ())))
       (#.$left$  (if (neq dir 'd) (setq dir 'g)))
       (#.$right$ (if (neq dir 'g) (setq dir 'd)))
       (#.$down$  (if (neq dir 'h) (setq dir 'b)))
       (#.$up$    (if (neq dir 'b) (setq dir 'h)))
       (#^ C (exit vdt))
       (nil)
       (t (tybeep)))
    ; on efface la queue
    (setq ochemin chemin)
    (tyco ( (car chemin) 128)
          (// (nextl chemin) 128)
          #/  )
    ; on calcule la nouvelle position
    (selectq dir
       (g (decr x) (when (<= x 0) (exit vdt)))
       (d (incr x) (when (>= x $xmax$) (exit vdt)))
       (h (decr y) (when (<= y 1) (exit vdt)))
       (b (incr y) (when (>= y $ymax$) (exit vdt)))
       (t ()))
    ; grandit-il ?
    (when (= ( score 20) 0)
          (newl chemin (car ochemin))
          (tycursor 60 0)
          (tyon (incr taille)))
    ; test si le lombric se coupe ou se cogne
    (setq xy (+ (* y 128) x))
    (when (or (memq xy chemin) (memq xy lobst))
          (exit vdt))
    ; on l'ajoute au chemin sans vraiment "conser"
    (nconc chemin (rplac ochemin xy ()))
    ; et on l'affiche
    (tyco x y #/@)
    ; affiche le score la aussi sans "conser"
    (when (>= (incr score) 1000)
          (setq score 0)
          (tycursor 50 0)
          (tyon (incr mscore)))
    (tycursor 53 0)
    (tyon score)
    ; l'attente programmee
    (when (< taille 150)
```

```
                    (repeat 50)
                    (when (< taille 100)
                          (repeat 100)
                          (when (< taille 50)
                                (repeat 200)
                                (when (< taille 20)
                                      (repeat 500)
                                      (when (< taille 10)
                                            (repeat 1000))))))

          ))
          ; pour ne pas effacer le score.
          (tyco 0 (1- $ymax$) '#"Vous vous etes tamponne."))
          'vdt
)))))))))

(de tyon (n)
    ; affiche la valeur de "n" sur 3 chiffres
    ; sans 'conser'.
    (tyo (+ #/0 (// n 100)))
    (setq n ( n 100))
    (tyo (+ #/0 (// n 10)) (+ #/0 ( n 10)))))

(de vdtend ()
    (mapc 'remfn '(vdt tyon vdtend))
    (autoload vdt vdt))
```

11.4 Le paragrapheur de fonctions

```
;================================================================
;
;  Le Lisp 80 v12  :  PRETTY  :  le paragrapheur interprété
;
;================================================================
;  (c) 1982    Jérôme CHAILLOUX   et   I.N.R.I.A.
;              B.P. 105, 78153 Le Chesnay Cédex France
;================================================================


;-----  Les fonctions internes

(de  p-p (l)
     ; paragraphe l'expression "l"
     (cond
       ((null l) (princn #/( ) (princn #/) ))
       ((atom l) (prin l))
       ((and (eq (car l) quote) (null (cddr l)))
          ; le cas de (QUOTE s)  =>  's
          (princn #/' )
          (p-p (cadr l)))
       ((and (consp (car l)) (eq (caar l) lambda))
          ; le cas ((LAMBDA ...) ...)  =>  (LET ...)
          (p-p (mcons 'let
                  (mapcar 'list (cadar l) (cdr l))
                  (cddar l))))
       ((p-inlinep l) (p-l l))
       (t (princn #/( )
          (let ((f (car l)) (l (cdr l)))
               (p-p f)
               (selectq (if (symbolp f)
                            (ptype f)
                            t)
                    (1 (p-progn))
                    (2 (p-pl) (p-progn t))
                    (3 (p-pl) (p-pl) (p-progn t))
                    (4 (p-cond))
                    (5 (p-pl) (p-cond))
                    (6 (bind ((lmargin (+ (lmargin) 5)))
                             (while l (p-pl) (p-pl) (when l (terpri)))))
                    (t (p-progn)))
               (and
                  l
                  (princn #/  )
                  (princn #/. )
                  (princn #/  )
                  (prin l)))
          (princn #/) ))))))
```

```
(de  p-pl ()
     ; edite l'element suivant (sauf le 1er)
     (bind ((lmargin (p-lmargin f)))
           (princn #/  )
           (p-p (nextl l)))))))

(de  p-l (l)
     ; imprime "l" sur la ligne (et ca rentre)
     (princn #/( )
     (p-p (nextl l))
     (while (consp l) (p-pl))
     (when l
          (princn #/  )
          (princn #/. )
          (princn #/  )
          (prin l))
     (princn #/) ))))

(de  p-progn (?)
     (if (and (null (cdr l)) (null ?))
         (p-pl)
         (bind ((lmargin (p-lmargin f)))
            (while (consp l)
                   (if (numberp (car l))
                   (p-pl)
                   (if (< (outpos) (lmargin))
                       (outpos (lmargin))
                       (terpri))
               (p-p (nextl l)))))))

(de  p-cond ()
     ; paragraphe le COND courant
     (bind ((lmargin (+ (lmargin) 3)))
          (while (consp l)
             (terpri)
             (if (p-inlinep (car l))
                 (p-l (nextl l))
                 (princn #/( )
                 (let ((l (nextl l)) (f T))
                      (p-p (nextl l))
                      (when l (p-progn t)))
                 (princn #/) )))))))

(de  p-lmargin (f)
     ; calcule la nouvelle marge gauche
     (+ (lmargin)
        (cond
           ((< (lmargin) 40)
               (if (< (plength f) 8)
                   (+ (plength f) 2)
                   4))
           ((< (lmargin) 60)
```

```
                2)
            (t 0))))))))

(de  p-inlinep (s)
     (let ((n (- (rmargin) (outpos))))
          (tag false (p-inlinepl s))
          (> n 0))))

(de  p-inlinepl (s)
     (cond
       ((numberp s)
          (when (< (decr n (length (explode s))) 0)
                (exit false)))
       ((atom s)
          (decr n (plength s))
          (when (<> (logand (ptype s) 128) 0)
                (decr n 2))
          (when (< n 0) (exit false)))
       ((and (eq (car s) quote) (null (cddr s)))
          (decr n)
          (p-inlinepl (cadr s)))
       (t (decr n 2)
          (while (consp s)
                 (p-inlinepl (nextl s))
                 (when (consp s) (decr n)))
          (when s (decr n 2) (p-inlinepl s))))))))
```

;----- Les fonctions utilisateur

```
(de  pprint (s)
     ; fonction utilisateur : paragraphe l'expression "s"
     (bind ((sprint T) (lmargin 0) (rmargin 76))
           (let ((f ())) (p-p s))
           (terpri))
  s)

(df pretty l
     ; paragraphe la liste de fonctions : "l"
     (mapc (lambda (l) (pprint (getdef l))
                       (terpri))
           l)))


(df prettyf (f . l)
     ; comme PRETTY mais le resultat est sorti sur fichier
     ; le nom du fichier est donne en 1er argument
     (output (filename f))
     (mapc (lambda (l) (pprint (getdef l))
                       (terpri))
```

```
            1)
        (output))))


(de prettyend ()
    (mapc 'remfn
          '(p-p p-pl p-l p-progn p-cond p-lmargin p-inlinep p-inlinepl))
    (autoload pretty pprint pretty prettyf)
    'prettyend)


;-----  Pose des ptypes prédéfinis manquants


(while (setq x (read))
   (ptype x (read)))

progn   1

lambda  2
let     2
lmargin 2
map     2
rmargin 2
tag     2
unless  2
untilexit 2
when    2

de      3
df      3
dm      3
dmc     3

cond    4

selectq 5

set     6
setq    6
```

11.5 L'éditeur video

```
;=============================================================
;
;    Le Lisp 80 v12  :   PEPE   -   (P)epe (E)st (P)resque (E)macs
;
;=============================================================
;   (c) 1982     Jérôme CHAILLOUX    et    I.N.R.I.A.
;                B.P. 105, 78153 Le Chesnay Cédex France
;=============================================================

; PEPE implémente une version réduite du célèbre éditeur Emacs
; dont il garde les caractéristiques suivantes :
;    - écrit entièrement en Lisp
;    - extensible
;    - indépendance vis-à-vis du matériel
;    - indépendance vis-à vis de le représentation des lignes
; en revanche il ne possède pas de reaffichage asynchrone.

;          *****    Que PEPE soit avec vous    *****


;------    Organisation des fichiers de PEPE :

;          PEPE.LL          la racine du mal ...
;          PEPTOP.LL        la boucle principale
;          PEPLNx.LL        les differentes gestions des lignes tassees
;          PEPMSC.LL        un ensemble de fonctions auxiliaires
;          PEPCMD.LL        les fonctions des commandes simples
;          PEPESC.LL        les fonctions des commandes ESCapes
;          PEPINI.LL        les affectations initiales des cles
;          PEPATO.LL        les fonctions autoload de PEPE

;------    Les variables globales de PEPE,
;          en plus de $xmax$ $ymax$ (dans VIRTTY) :

;          $buffer$       la liste des lignes
;          $file$         nom du fichier du buffer
;          $ydisplay$     no de la lere ligne de la fenetre visible
;          $xcursor$      la colonne courante du curseur
;          $ycursor$      la ligne courante du curseur
;          $modbuf$       indicateur, vrai si le tampon n'a pas ete sauve
;          $column$       la colonne virtuelle courante
;          $commands$     la A-Liste des cles normales
;          $escommands$   la A-Liste des commandes escape x
;          $kill$         la derniere ligne detruite par ^ T

;------    La fonction AUTOLOAD qui charge le reste
```

```
(dm pepe 1
    (load PEPTOP)
    (load PEPLN2)
    (load PEPMSC)
    (load PEPCMD)
    (load PEPESC)
    (load PEPINI)
    1))

;-----  Le récuperateur de l'espace de PEPE

(df pepend ()
    (mapc 'remfn
          '(
          pkldisplay pklprint pkllength pklbreak
          pkljoin pklchar pklcharn  pklinschar
          pkldelchar pklimplode pklexplode          ; dans PEPLN2

          deadend currentline cursor
          redisplayhome redisplay
          bolp eolp bobp eobp bosp eosp             ; dans PEPMSC

          left right endline begline up down
          nextscreen insertchar deletechar breakline
          joinlines killline                        ; dans PEPCMD

          pepefile                                  ; dans PEPTOP

          escommand readname
          readfile writefile                        ; dans PEPESC

          defkey defesckey                          ; dans PEPINI

        ))
      (mapc 'makunbound
          '($buffer$ $file$ $commands$ $escommands$))

      (autoload pepe pepe pepend)
      'pepend)))

;=================================================================
;
;   Le Lisp 80 v12  :   PEPE   -   PEPTOP  la boucle principale
;
;=================================================================

;-----  les appels de PEPE
```

```
(df pepe (f)
    ; la forme FSUBRee de la fonction suivante
    (pepefile f)))

(de  pepefile (f)
     ; la fonction qui evalue son nom de fichier
     (when f
           (setq $buffer$ ()
                 $file$ 'tmp
                 $xcursor$ 0 $ycursor$ 0
                 $column$ 0 $ydisplay$ 0
                 $modbuf$ ())
           (cond
               ((eq f T)
                   ; je veux un fichier scratch
                   (setq $buffer$ (cons (pklimplode ()) ())))
               ((symbolp f)
                   ; ce doit etre un fichier qui existe
                   (tag eoc (readfile f))
                   (setq $file$ f))
               ((consp f)
                   ; c'est une liste de fonctions
                   (flet ((eol ()
                               (let ((i 0) (r ()))
                                    (repeat (outpos) (newr r (outbuf i))
                                            (outbuf i #/ )
                                            (incr i))
                                    (outpos 0)
                                    (repeat (lmargin) (outbuf (outpos) #/ )
                                                      (outpos (1+ (outpos))))
                                    (newl $buffer$ (pklimplode r)))))
                          (eval f))
                    (setq $buffer$ (nreconc $buffer$ ())))
               (t (syserror 'pepe '| mauvais argument | f))))
    (tyattrib ())
    (redisplay)
    ;
    ; le top-level proprement dit de PEPE
    ;
    (let ((lcmd))
     ; lcmd est la liste des commandes non encore executees
     (flet ((iteval (n) (newr lcmd n)))
       (untilexit pepe
         (tag eoc
            (tycursor $xcursor$ $ycursor$)
            (let ((c (or (nextl lcmd) (tyi))))
                 (let ((l (cassq c $commands$)))
                      (cond
                          ((null l)
                               ; ce n'est pas une commande
                               (if (< c 32)
                                   (deadend)
```

```
                                    (insertchar c)))
                            ((symbolp l)
                                ; c'est une commande
                                (apply l))
                            (t  ; egalement
                                (eprogn l)))))))
      ; je sors de PEPE : qu'Il soit avec vous
      (tycursor 0 $ymax$)
      (tycleol)
      (tycursor 0 #.(1- $ymax$))
      (tycleol)
      'PEPE)))))))

;================================================================
;
;  Le Lisp 80 v12  :  PEPE  -  PEPLN1  :  gestion des lignes (1)
;
;================================================================


;*****   1er modele de gestion des lignes :
;       une ligne n'est qu'une liste de code ASCII.
;       C'est TRES facile a gerer mais l'occupation
;       memoire est prohibitive.
;
;       C'est la meilleure maniere de decrire les 11 fonctions
;       de gestion des lignes (et cela permet de mettre au point
;       les gestions plus sophitiquees comme PEPLN2 puis PEPLN3)


(de  pkldisplay (pkl)
     ; affiche la ligne tassee "pkl"
     (tyco 0 $ycursor$ pkl))

(de  pklprint (pkl)
     (mapc 'princn pkl)
     (terpri))

(de  pkllength (pkl)
     ; retourne la taille d'une ligne tassee
     (length pkl))

(de  pklbreak (pkl n)
     ; casse la ligne tassee "pkl" a la position "n"
     ; retourne un CONS  (pkl-gauche . pkl-droite)
     (cond
        ((= n 0) (cons () pkl))
        ((> n (pkllength pkl)) (cons pkl ()))
        (t (cons pkl (let ((l (nthcdr (1- n) pkl)))
                          (progl (cdr l) (rplacd l () ))))))))))
```

```
(de  pkljoin (pkl1 pkl2)
     ; retourne la ligne composee des lignes pkl1 et pkl2
     (nconc pkl1 pkl2)))

(de  pklchar (pkl n)
     ; retourne le nieme caractere de la ligne pkl
     (or (nth n pkl) #/  )))

(de  pklcharn (pkl n)
     ; retourne les "n" derniers caracteres de "pkl"
     (nthcdr n pkl))

(de  pklinschar (pkl n c)
     ; insere dans "pkl" le caractere "c" a la position "n"
     ; ne change pas l'adresse physique de "pkl"
     (cond
        ((= n 0) (rplac pkl c (cons (car pkl) (cdr pkl))))
        ((> n (pkllength pkl)) (nconc1 pkl c))
        (t (let ((l (nthcdr n pkl)))
                (rplac l c (cons (car l) (cdr l))))))))

(de  pkldelchar (pkl n)
     ; enleve dans "pkl" le caractere a la position "n"
     ; ne change pas l'adresse physique de "pkl"
     (cond
        ((= n 0)
            (if (cdr pkl)
                (rplac pkl (cadr pkl) (cddr pkl))
                (rplacd pkl ())))
        ((> n (pkllength pkl)) ())
        (t (let ((l (nthcdr (1- n) pkl)))
                (rplacd l (cddr l)))))))

(de  pklimplode (l)
     ; transforme la liste de caracteres "l" en
     ; une ligne packee "pkl"
     l))

(de  pklexplode (pkl)
     ; transforme la ligne packee "pkl"
     ; en une liste de caracteres "l"
     pkl))))
```

```
;=============================================================
;
;  Le Lisp 80  :  PEPE  -  PEPLN2  :  gestion des lignes (2)
;
;=============================================================
```

```
;*****   2eme modele de gestion des lignes :
;       une ligne est un CONS :  (n . l)
;       dans lequel "n" est le nombre d'espaces en tete de ligne
;       et "l" les autres caracteres de la ligne
;       La gestion est un peu plus lourde que precedement
;       mais le gain est tres appreciable specialement pour
;       des programmes ecrits en Lisp.
;       Voici les 11 fonctions de gestion des lignes

(de pkldisplay (pkl)
    ; affiche la ligne tassee "pkl" a la position $ycursor$
    (tyco (car pkl) $ycursor$ (cdr pkl)))

(de pklprint (pkl)
    ; imprime (sur fichier) la ligne packee "pkl"
    (repeat (car pkl) (princn #/  ))
    (mapc 'princn (cdr pkl))
    (terpri))))

(de pkllength (pkl)
    ; retourne la taille d'une ligne tassee
    (+ (car pkl) (length (cdr pkl))))

(de pklbreak (pkl n)
    ; casse la ligne tassee "pkl" a la position "n"
    ; retourne un CONS  (pkl-gauche . pkl-droite)
    (if (>= n (car pkl))
        ; dans le texte
        (cons pkl
              (let ((l (nthcdr (- n (car pkl)) pkl)))
                   (pklimplode (progl (cdr l) (rplacd l () )))))
        ; dans le renfoncement
        (cons [n] (rplaca pkl (- (car pkl) n)))))

(de pkljoin (pkl1 pkl2)
    ; retourne la ligne composee des lignes pkl1 et pkl2
    (pklimplode
        (append (pklexplode pkl1) (pklexplode pkl2)))))

(de pklchar (pkl n)
    ; retourne le nieme caractere de la ligne pkl
    (if (>= n (car pkl))
        ; dans le texte
        (or (nth (- n (car pkl)) (cdr pkl)) #/  )
        ; dans le renfoncement
        #/  ))))

(de pklcharn (pkl n)
    ; retourne les "n" derniers caracteres de pkl
    ; sous forme d'une liste de caracteres
    (nthcdr n (pklexplode pkl))))
```

```
(de pklinschar (pkl n c)
    ; insere dans "pkl" le caractere "c" a la position "n"
    ; ne change pas l'adresse physique de "pkl"
    (cond
       ((>= n (pkllength pkl))
        ; a la fin de la ligne
        (nconc pkl [c]))
       ((> n (car pkl))
        ; dans le texte
        (let ((l (nthcdr (- n (car pkl)) (cdr pkl))))
             (rplac l c (cons (car l) (cdr l)))))
       (t
        ; dans le renfoncement
        (if (= c #/ )
            (rplaca pkl (1+ (car pkl)))
            (rplac pkl n (cons c
                  (progn (repeat (- (nextl pkl) n)
                            (newl pkl #/ ))
                         pkl)))))))))))

(de pkldelchar (pkl n)
    ; enleve dans "pkl" le caractere a la position "n"
    ; ne change pas l'adresse physique de "pkl"
    (if (>= n (car pkl))
        ; je suis dans le texte
        (let ((l (nthcdr (- n (car pkl)) pkl)))
             (ifn (cddr l)
                  ; dernier caractere de la ligne
                  (rplacd l ())
                  ; au milieu du texte
                  (nextl l)
                  (displace l (cdr l))))
        ; je suis dans le renfoncement
        (rplaca pkl (1- (car pkl)))))))))

(de pklimplode (l)
    ; transforme la liste de caracteres "l" en
    ; une ligne packee "pkl"
    (let ((n 0))
         (while (and l (eq (car l) #/ ))
                (incr n)
                (nextl l))
         (cons n l)))))))

(de pklexplode (pkl)
    ; transforme la ligne packee "pkl"
    ; en une liste de caracteres "l"
    (let ((n (car pkl)) (l (cdr pkl)))
         (repeat n (newl l #/ ))
         l)))))
```

```
;=============================================================
;
;  Le Lisp 80 v12  :  PEPE  -  PEPMSC  fonctions auxiliaires
;
;=============================================================

(de  deadend ()
     ; fin de la route : on ne plus plus bouger
     (tybeep)
     (exit eoc))

(de  currentline ()
     ; retourne la ligne courante
     (nth (+ $ydisplay$ $ycursor$) $buffer$))

(de  cursor (x y)
     ; change la position du curseur de PEPE
     (setq $xcursor$ x $ycursor$ y)
     (tycursor x y))

(de modbuffer ()
    ; positionne l'indicateur  <M>
    (unless $modbuf$
            (setq $modbuf$ t)
            (redisplay $ymax$)
            (tycursor $xcursor$ $ycursor$)))

(de  redisplayhome ()
     ; reaffiche tout l'ecran et se positionne en haut
     (setq $xcursor$ 0 $ycursor$ 0 $column$ 0)
     (redisplay))

(de  redisplay (n)
     ; reaffiche toute la fenetre visible
     ; a partir de 'n' (optionnel)
     (tycursor 0 (or n $ycursor$))
     (let (($ycursor$
              (if (and (numberp n) (tycleos))
                  n
                  (tycls)
                  0)))
          (let ((b (nthcdr (+ $ydisplay$ $ycursor$) $buffer$))
                ($xcursor$ 0))
               (repeat (- #.$ymax$ $ycursor$)
                  (when b
                        (pkldisplay (nextl b))
                        (incr $ycursor$)))))
    (tycursor 0 #.$ymax$)
    (tyattrib t)
    (tyo '#"Pepe : " (explode $file$) '#"  "
         (explode (length $buffer$)) '#" ligne")
    (when (> (length $buffer$) 1) (tyo #/s))
```

```
      (when $modbuf$ (tyco 32 #.$ymax$ '#"<M>"))
      (tyattrib ()))


;-----   une brouette de prédicats utiles

(de  bolp ()
      ; teste si on se trouve en debut de ligne
      (= $xcursor$ 0)))

(de  eolp ()
      ; teste si on se trouve en fin de ligne
      (>= $xcursor$ (pkllength (currentline))))

(de  bobp ()
      ; teste si on se trouve en debut de buffer
      (and (= $ydisplay$ 0) (= $ycursor$ 0)))

(de  eobp ()
      ; teste si on se trouve en fin de buffer
      (>= (+ $ydisplay$ $ycursor$ 1) (length $buffer$)))

(de  bosp ()
      ; teste si on se trouve au debut de l'ecran
      (= $ycursor$ 0))

(de  eosp ()
      ; teste si on se trouve a la fin de l'ecran
      (>= $ycursor$ #.(1- $ymax$))))))



;=============================================================
;
;  Le Lisp 80 v12  :  PEPE  -  PEPCMD  les commandes de PEPE
;
;=============================================================


(de  left ()
      ; un coup a gauche
      (ifn (bolp)
            (setq $column$ (decr $xcursor$))
            (up)
            (endline)))

(de  right ()
      ; un coup a droite
      (ifn (eolp)
```

```
(de  endline ()
     ; va a la fin de la ligne
     (setq $xcursor$ (pkllength (currentline))
           $column$ $xcursor$))

(de  begline ()
     ; va au debut de la ligne
     (setq $xcursor$ 0 $column$ 0)))

(de  up ()
     ; va la ligne precedente
     (if (bobp)
         (deadend)
         (ifn (bosp)
              (setq $ycursor$ (1- $ycursor$)
                    $xcursor$ (min $column$
                                   (pkllength (currentline))))
              (decr $ydisplay$ 6)
              (incr $ycursor$ 5)
              (when (< $ydisplay$ 0)
                    (setq $ydisplay$ 0 $ycursor$ 0))
              (redisplay))))))

(de  down ()
     ; va a la ligne suivante
     (if (eobp)
         (deadend)
         (ifn (eosp)
              (setq $ycursor$ (1+ $ycursor$)
                    $xcursor$ (min $column$
                                   (pkllength (currentline))))
              (incr $ydisplay$ 6)
              (decr $ycursor$ 5)
              (redisplay)))))

(de  nextscreen ()
     ; passe a l'ecran suivant
     (when (> (+ $ydisplay$ #.(1- $ymax$)) (length $buffer$))
           (deadend))
     (incr $ydisplay$ #.(1- $ymax$))
     (redisplayhome))

(de  prevscreen ()
     ; passe a l'ecran precedent
     (when (< (decr $ydisplay$  #.(1- $ymax$)) 0) (setq $ydisplay$ 0))
     (redisplayhome))

(de  insertchar (c)
     ; rajoute le caractere "c" a la position courante
     (let ((n (pkllength (currentline))))
          (cond
             ((>= n $xmax$) (deadend))
```

```
            ((= n 0) (rplaca (nthcdr (+ $ydisplay$ $ycursor$) $buffer$)
                             (pklimplode [c])))
            (t
               (pklinschar (currentline) $xcursor$ c)))
          #.(if (tyinsch 12)
                '(tyinsch c)
                '(tyo (pklcharn (currentline) $xcursor$)))
          (setq $column$ (incr $xcursor$))
          (modbuffer)))))))

(de  deletechar ()
     ; enleve le caractere a la position du curseur
     (if (eolp)
         (joinlines)
         (pkldelchar (currentline) $xcursor$)
         #.(if (tydelch)
               '(tydelch)
               '(tyo (pklcharn (currentline) $xcursor$) 32))
         (modbuffer)))))

(de  breakline ()
     ; casse la ligne a la position du curseur
     (let ((l (nthcdr (+ $ycursor$ $ydisplay$) $buffer$)))
          (ifn (eolp)
               (let ((ll (pklbreak (car l) $xcursor$)))
                    (rplac l (car ll) (cons (cdr ll) (cdr l))))
               (if (eobp)
                   (newr $buffer$ (pklimplode ()))
                   (rplacd l (cons (pklimplode ()) (cdr l)))))
          (modbuffer)
          (redisplay $ycursor$))))

(de  joinlines ()
     ; accroche la ligne suivante a la courante
     (let ((l (nthcdr (+ $ycursor$ $ydisplay$) $buffer$)))
          (rplac l (pkljoin (car l) (cadr l)) (cddr l))
          (modbuffer)
          (redisplay $ycursor$)))

(de  killline ()
     ; detruit la ligne courante
     (if (bobp)
         (if (cdr $buffer$)
             (nextl $buffer$)
             (setq $buffer$ [(pklimplode ())]))
         (let ((l (nthcdr (+ $ydisplay$ (1- $ycursor$)) $buffer$)))
              (rplacd l (cddr l))))
     (modbuffer)
     (if (eobp)
         (or (bobp) (up))
         ; et l'autre cas si la ligne est plus petite
         (when (< (pkllength (currentline)) $xcursor$)
```

```
                (endline)))
      #.(if (tydelln) '(progn (tydelln) (redisplay (- $ymax$ 2)))
                      '(redisplay $ycursor$))))))


;=============================================================
;
;  Le Lisp 80 v12  :  PEPE  -  les commandes de type <esc> X
;
;=============================================================


;-----   L'interprete des commandes <esc>
;       la A-Liste des commandes se trouve dans :escommand:

(de  escommand ()
     (let ((c (or (nextl lcmd) (tyi))))
          ; passage en majuscule
          (when (and (>= c #/a) (<= c #/z))
                (decr c 32))
          (let ((l (cassq c $escommands$)))
               (cond
                  ((null l)
                     ; ce n'est pas une commande <esc>
                     (deadend))
                  ((symbolp l)
                     ; c'est une commande
                     (apply l))
                  ((numberp (car l))
                     ; c'est une action autoload
                     (when (input (filename 'pepato))
                           (repeat (1- (car l)) (read))
                           (eval (progl (read) (input ()) (teread)))))
                  (t ; egalement
                     (eprogn l))))))


;-----   Lecture d'un nom sur le terminal

(de  readname ()
     (tyco 40  #.$ymax$ '#" Nom ")
     (read))

;-----   Lecture d'un fichier

(de  readfile (f)
     (let ((l) (c))
          (setq $buffer$ ()
                $xcursor$ 0
                $ycursor$ 0
                $ydisplay$ 0)
```

```
        (unless (input (filename f))
                (setq $buffer$ [(pklimplode ())])
                (input)
                (deadend))
        (inmax 0) (readcn)
        (untilexit eof
             (setq c (readcn))
             (cond ((= c #^ M)    ; rc
                        (while (and (consp l) (= (car l) #/  ))
                               (nextl l))
                        (newl $buffer$ (pklimplode (nreconc l ())))
                        (setq l ()))
                   ((< c 32))    ; cntrl-car
                   (t (newl l c)))))
        (setq $buffer$ (nreconc $buffer$ ()))
        (input))

;-----   Ecriture d'un fichier

(de  writefile (f)
     (unless (output (filename f))
             (output)
             (deadend))
     (let (($xcursor$ $xcursor$)
           ($ycursor$ $ycursor$))
          (cursor 33 $ymax$)
          (bind ((rmargin 80))
                (mapc 'pklprint $buffer$)))
     (output)
     (setq $modbuf$ ())
     (redisplay $ymax$))


;-----   verificateur de parentheses a la Lisp

(de matchparent ()
    (let ((l (pklcharn (currentline) $xcursor$))
          (n 0))
         (decr $xcursor$)
         (untilexit fin
            (selectq (matchparenc)
                (#/( (incr n))
                (#/) (when (= n 1) (right) (exit fin))
                     (when (<> n 0) (decr n)))
                (#/# (selectq (matchparenc)
                        (#// (matchparenc))
                        (#/" (until (= (matchparenc) #/")))
                        (t ())))
                (#/; (setq l ()))
                (t ())))))
```

```
(de matchparenc ()
    ; retourne le caractere suivant
    (incr $xcursor$)
    (or (nextl l)
        (progn (begline)
               (down)
               (setq l (pklexplode (currentline)))
               (nextl l))))
```

```
;=============================================================
;
;  Le Lisp 80 v12  :  PEPE  -  PEPINI  initialisation des clés
;
;=============================================================
```

;----- Initialisation des variables globales

```
(setq $buffer$ (cons (pklimplode ()) ()))

(setq $xcursor$ 0 $ycursor$ 0 $column$ 0 $ydisplay$ 0)

(setq $file$ 'TMP $commands$ () $escommands$ () $modbuf$ () )

(df defkey (k . f)
      (newl $commands$
            (cons (eval k) (if (symbolp (car f))
                               (car f)
                               f)))))
```

;----- initialisation standard (à la EMACS)

```
(progn

(defkey  #^ A      begline)
(defkey  $left$   left)
(defkey  #^ C      (exit pepe))
(defkey  #^ D      deletechar)
(defkey  #^ E      endline)
(defkey  #^ L      redisplay)
(defkey  $right$  right)
(defkey  #^ M      (breakline) (right))
(defkey  $down$   down)
(defkey  #^ O      breakline)
(defkey  $up$     up)
(defkey  #^ T      killline)
(defkey  #^ V      nextscreen)
```

```
(defkey  #$7F     (left) (cursor $xcursor$ $ycursor$) (deletechar))
(defkey  27       escommand)

)

(df defesckey (k . f)
      (newl $escommands$
            (cons k (if (symbolp (car f))
                        (car f)
                        f)))))

(progn

(defesckey  #/E    2)
(defesckey  #/F    (setq $file$ (readname)) (redisplay))
(defesckey  #/I    3)
(defesckey  #/R    (readfile (setq $file$ (readname))) (redisplay))
(defesckey  #/S    (writefile $file$))
(defesckey  #/V    prevscreen)
(defesckey  #/W    (writefile (readname)) (redisplay))
(defesckey  #/Z    (writefile $file$)
                   (tycursor 0 $ymax$)
                   (loadfile $file$)
                   (exit pepe))
(defesckey  #/?    1)
(defesckey  #/)    matchparent)
(defesckey  #/<    (setq $ydisplay$ 0) (redisplayhome))
(defesckey  #/>    (setq $ydisplay$ (- (length $buffer$)
                                       #.(// $ymax$ 2)))
                   (when (< $ydisplay$ 0) (setq $ydisplay$ 0))
                   (redisplayhome))
)
)))))))


;=============================================================
;
;   Le Lisp 80 v12  :  PEPE  les fonctions autoload
;
;=============================================================


;*******     Attention : ne pas modifier l'ordre de ces fonctions
;           voir PEPINI.LL dans lequel cet ordre est utilise

;*****  (1)   Affiche le recapitulatif des commandes
;             la documentation "en-line" de PEPE

(progn
    (tycls)
    (tycursor 0 0)
    (input (filename 'PEPHLP))
```

```
      (untilexit eof (tyo (readcn)))
      (input ())
      (until (tyi))
      (redisplay))


;***** (2)   Evaluation de tout le tampon sans sortir de PEPE
;           dependant du format des lignes

(let ((f $buffer$))
      (flet ((bol ()
                  (ifn f
                        (exit eoc)
                        (let ((l (cdr (nextl f))) (n -1))
                             (ifn l
                                  (bol)
                                  (while l
                                        (inbuf (incr n) (nextl l)))
                                  (inbuf (incr n) 13)
                                  (inbuf (incr n) 10)
                                  (inmax n))))))
            (untilexit eoc
                       (print '|=> | (eval (read))))))


;***** (3)  insere un nouveau fichier au niveau
;           de la ligne courante.

(let ((l (nthcdr (+ $ydisplay$ $ycursor$) $buffer$))
      ($buffer$)
      ($xcursor$)
      ($ycursor$)
      ($ydisplay$))
     (readfile (readname))
     (displace l (nconc $buffer$ (cons (car l) (cdr l)))))
     (redisplay $ycursor$))
```

11.6 Pistage, Mise–au–point et Pas à pas

```
;=============================================================
;
;              Le Lisp 80 v12  :   TRACE et UNTRACE
;
;=============================================================
;  (c) 1982    Jérôme CHAILLOUX   et   I.N.R.I.A.
;              B.P. 105, 78153 Le Chesnay Cédex France
;=============================================================

;       La variable globale TRACE contient
;       la liste des fonctions pistées.

(setq trace ())

;-----   Les fonctions de pistage


(df trace l
    ; trace la liste de fonctions 'l
      (mapc (lambda (f)
            (let ((fval (valfn f)))
                 (ifn (typefn f)
                      (print '|je ne connais pas | f)
                      (putprop f fval 'trace)
                      (unless (memq f trace) (newl trace f))
                      (valfn f
                             [(car fval)
                                ['prin (kwote f) ''| ---> | ]
                                ['mapc (lambda (trace)
                                         (prin trace '| = |
                                               (cval trace) '|  |))
                                         (kwote (flat (car fval))) ]
                                '(terpri)
                                ['print
                                    (kwote f)
                                    ''| <--- |
                                    ['progn   (cdr fval)]]]))))
            l)
      l)))

(de flat (l r)
    ; retourne la liste de tous les atomes
    ; de la liste l
    (cond ((consp l) (flat (car l) (flat (cdr l) r)))
          (l (cons l r))
          (t r)))
```

```
(df untrace l
      ; enleve la trace de toutes les fonctions
      ; de la liste 'l' ou de toutes les fonctions
      ; tracees si 'l' = ()
      (mapc (lambda (f)
                (let ((fval (getprop f 'trace)))
                  (if (atom fval)
                      (print f '| n'etait pas tracee.|)
                      (valfn f fval)
                      (remprop f 'trace))))
            (or l (setq l trace trace ())))
      l)

(de tracend ()
    ; recupere la place de la TRACE
    (autoload trace trace untrace tracend)
    (setq trace ())
    'tracend)

;================================================================
;
;     Le Lisp 80 v12  :  la mise au point interactive
;
;================================================================
;  (c) 1982    Jérôme CHAILLOUX    et    I.N.R.I.A.
;              B.P. 105, 78153 Le Chesnay Cédex France
;================================================================


;-----    La fontion de mise au point

(setq debug ())

(df debug  l
    (setq debug l)
    (eprogn l) )

;-----    Récupération du BREAK

(de break ()
    (when debug
          (printstack 10)
          (bind ((prompt (concat '|>| (prompt))))
                (print '|vous etes dans la fonction BREAK.  ^ D pour sortir|)
                (untilexit break
                      (tag syserror (print '|= | (eval (read)))))
                'break))
    (exit syserror 'break))


;-----    Pour sortir du BREAK ...
```

```
(dmc ^ D () '(exit break))

;-----  Impression de la pile

(de printstack (m)
    (let ((n (length (cstack))))
         (mapc (lambda (l)
                       (outpos 3)
                       (prin (decr n))
                       (outpos 10)
                       (tag print
                            (flet ((eol () (flush) (exit print)))
                                  (prin (or (findfn (cadr l)) l))))
                       (terpri))
               (cddr (cstack m)))))))

;-----  Récupération de la place

(de debugend ()
    (mapc 'remfn '(debug printstack))
    (setq debug ())
    (autoload debug debug))

;===============================================================
;
;            Le Lisp 80 v12 :  STEP exécuteur pas à pas
;
;===============================================================
;  (c) 1983    Emmanuel Saint-James
;===============================================================



(de step (traceval deep step)
    (flet ((stepin (traceval)
                (incr deep)
                (ifn (eq (1+ step) deep)
                     traceval
                     (incr step)
                     (loopselect |=> |
                        (prindeep deep '| ---> | traceval)
                        (#/= (toplevel))
                        (#/< (exit |=> | traceval))
                        (#/  (exit |=> | (kwote (read))))
                        (#/. (exit |=> | (setq step 0) traceval))
                        (#/, (exit |=> | (setq step (eval (read)))
                                         traceval))
                        (#/> (exit |=> |
                                (list 'traceval (kwote traceval)) ))
                        (t (tyo 7)))))
           (stepout (traceval)
               (progl
```

```
                    (ifn (eq step deep)
                         traceval
                         (decr step)
                         (loopselect
                            |<= |
                            (prindeep deep '| <--- | traceval)
                            (#/= (toplevel))
                            (#/< #/>) (exit |<= | traceval))
                            (#/. (exit |<= | (setq step 0) traceval))
                            (#/, (exit |<= | (setq step (eval (read)))
                                                  traceval))
                            (t (tyo 7))))
                    (decr deep)))
        (setq deep 0 step 0)
        (traceval traceval)))

(dm loopselect loopselect
    (displace loopselect
        (list
            'bind
            (list (list 'prompt (kwote (cadr loopselect))))
            (list
                'untilexit
                (cadr loopselect)
                (mcons 'selectq
                       (caddr loopselect)
                       (cdddr loopselect))))))

(de prindeep (deep prindeep traceval)
    (prin deep)
    (outpos 4)
    (prin prindeep traceval)
    (flush)
    (tyo #.(+ 128 #/ ))
    (progl (princn (tyi)) (terpri)))
```

CHAPITRE 12

INDEX

Index des Fonctions

INDEX des concepts

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique